ESIREM
ÉCOLE SUPÉRIEURE D'INGÉNIEURS
NUMÉRIQUE ET MATÉRIAUX

École associée Polytech

Fundamentals of C++.

From **beginner** to **beyond.**

Last updated on October, 7, 2022 by dG

Lecture #01

User-defined Data Types

Slides are available on http://ginhac.com/teaching/ITC313/latest/01-usertypes.pdf

All code samples are available on directory “samples/latest/01-usertypes” from https://github.com/dginhac/esirem-itc313

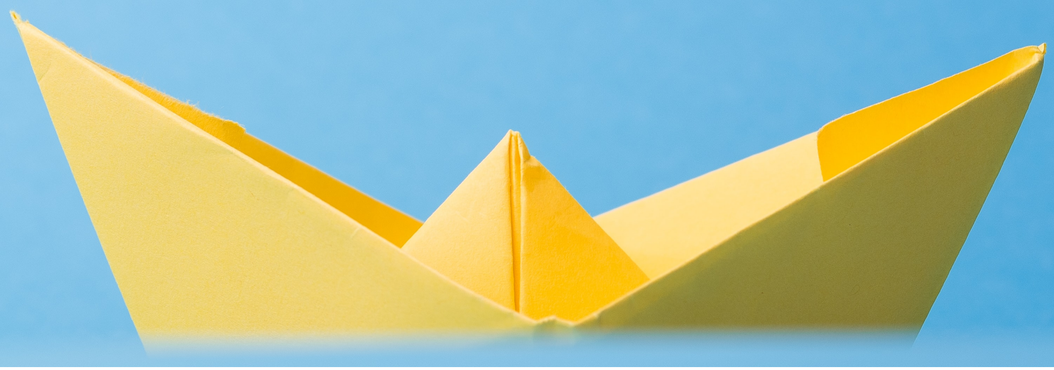


Photo by Alex Padurariu on Unsplash

Introduce the fundamentals of Object-Oriented Programming with the creation of simple classes and objects on concrete examples

Enjoy! 😉

Lecture #01
User-defined Data Types

Lecture #00
Course Introduction

Today

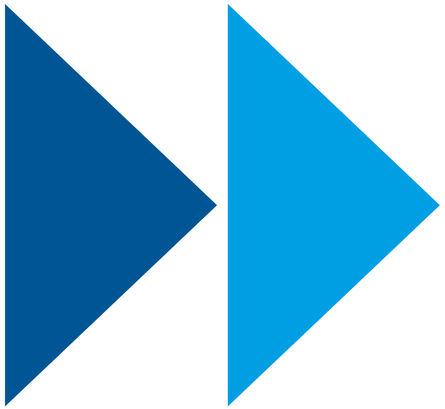
Lecture #02
Inheritance

Lecture #03
Polymorphism

Lecture #04
STL Containers

Lecture #05
Templates

Lecture #06
Exceptions



AGENDA

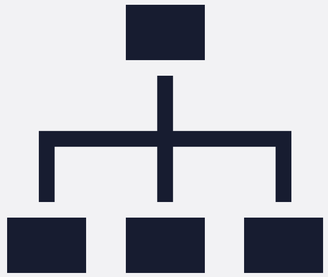
01 – Basics of Objects / Classes

02 – A realistic example of class

03 – More on functions

04 – Other user-defined types

05 – Organize your types in namespaces



C++ Type Classification

C++ is a strongly typed language

Variables can hold only certain types of values.

Variables must be declared before they're used and can't change type.

Fundamental types built into C++

Numbers, booleans, single characters.

User-defined types in libraries and in your programs

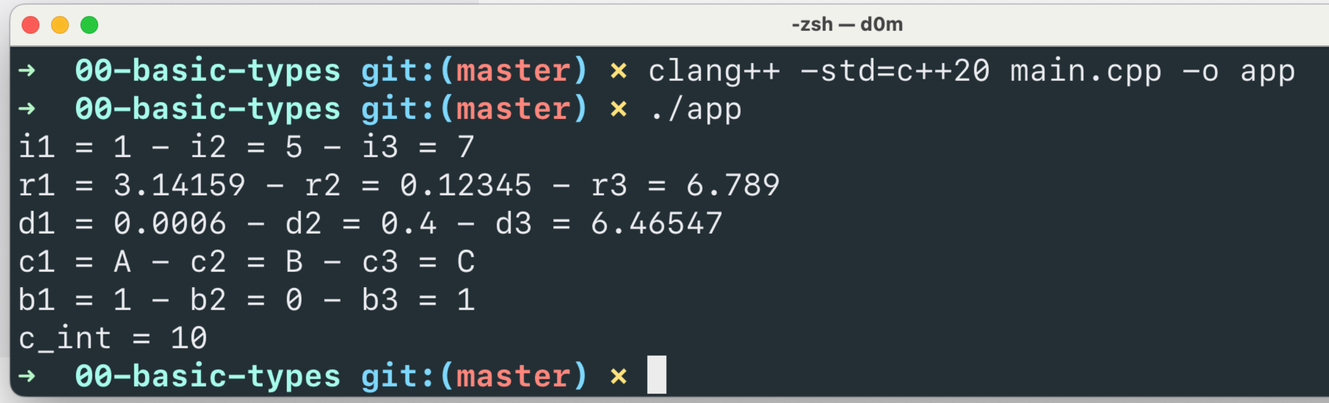
Classes, Structures, Enumerations, Unions.

References, pointers.

Reminder on basic types

00-basic-types/main.cpp

```cpp
// integer type can be short, long, and unsigned
int i1 = 1; // expression from C-language
int i2(5); // expression list from C++ Constructor
int i3{7}; // initializer list since C++11

// floating point type (simple or double precision)
float r1 = 3.14159, r2(0.12345), r3{6.789};
double d1 = 6e-4, d2(0.4), d3{6.46546764};

// character type
char c1 = 'A', c2('B'), c3{'C'};

// boolean : true or false
bool b1 = true, b2(false), b3{true};

// constant can not be modified and
// must be initialized when declared
const int c_int = 10;
```

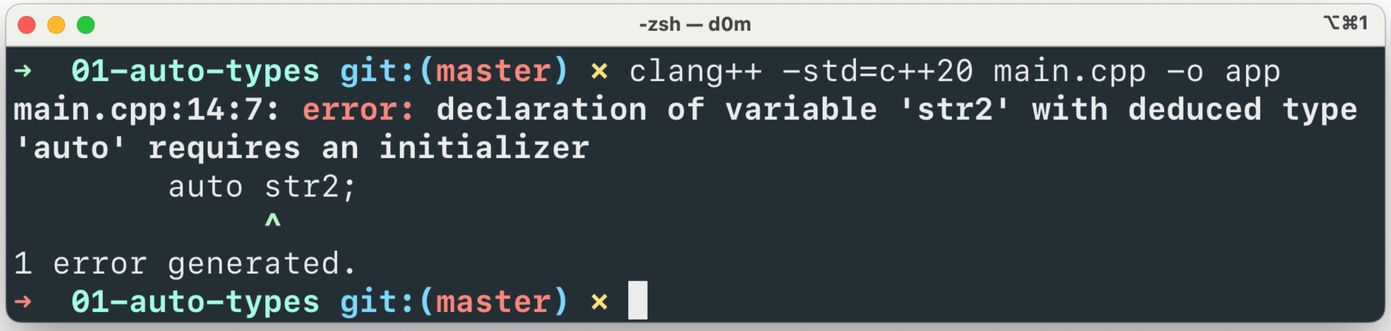


Auto in C++

The auto keyword in C++ automatically detects and assigns a data type to the variable with which it is used. The compiler analyses the variable's data type by looking at its initialization.

01-auto-types/main.cpp

```cpp
// variables can be defined without initialization
int i0;
// auto variables must be initialized to infer the type
auto i1 = 1;
auto r1 = 3.14159;
auto c1 = 'A';
auto str1 = "hello, world";
// generate a compilation error
auto str2;
```



Best practices

Always initialize variables.
Use auto and C++11 or more compiler.

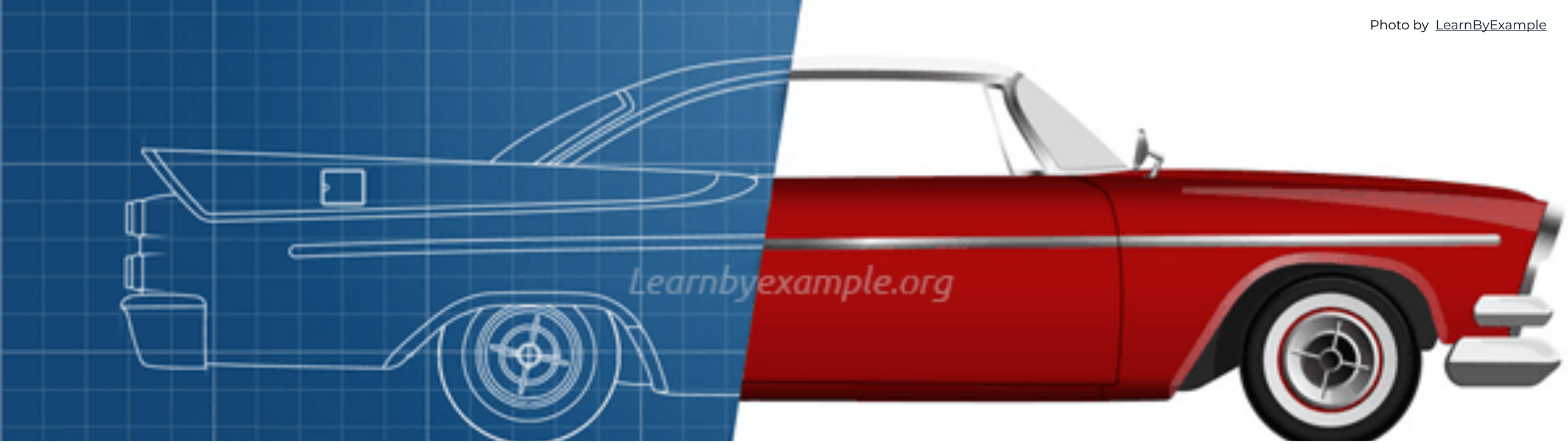


Photo by LearnByExample

User-defined types

To grapple with complex problems, you need to create
precise representations of the data that you are talking about.
The closer these representations correspond to reality, the easier it is to write the program.

In C++, the most workable representations are **classes** and **objects**.

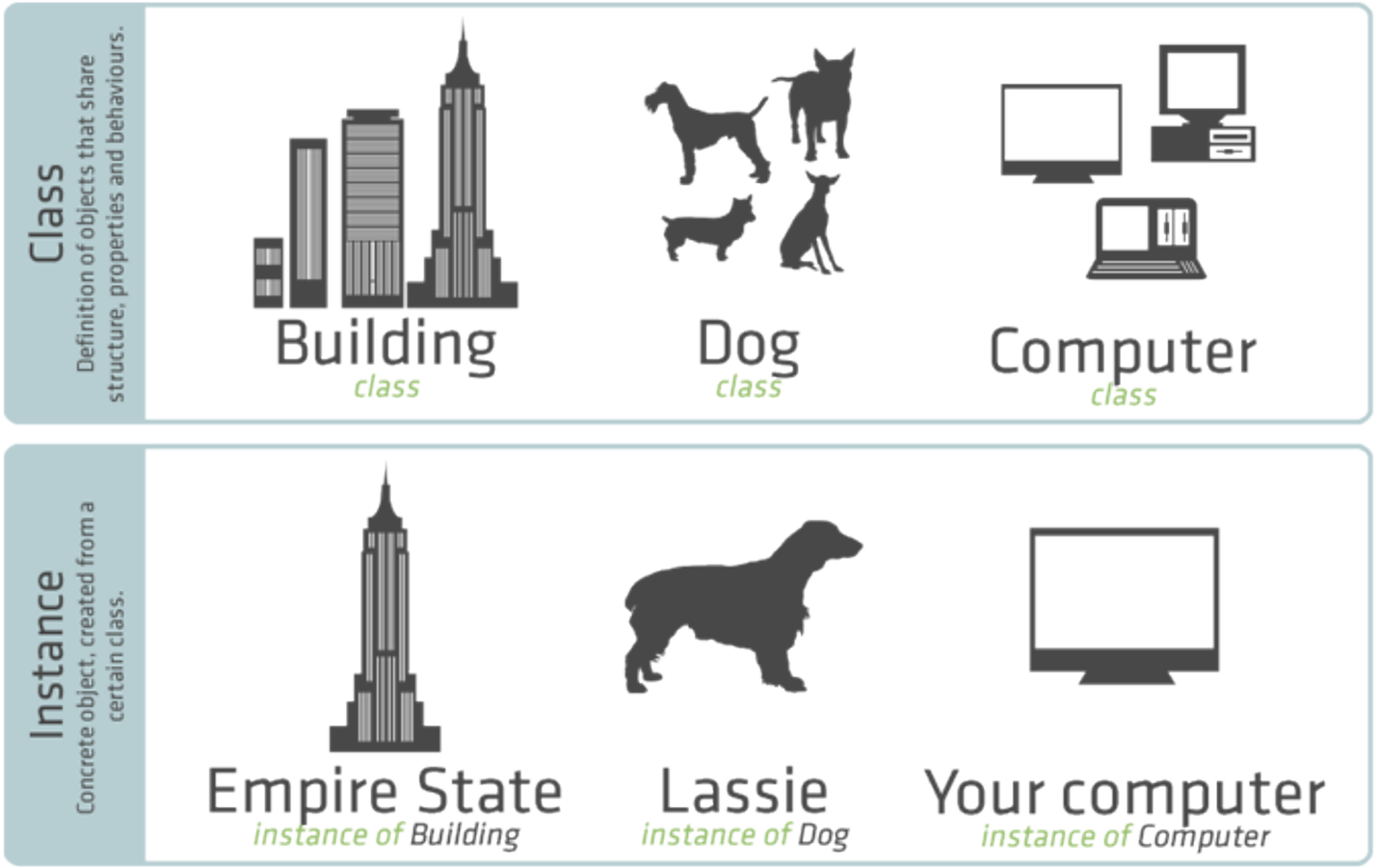
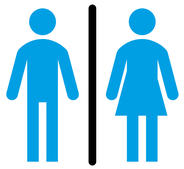


Illustration by Sencha

class: model for a new type of objects represented by a collection of variables combined with a set of related functions.

object: single instance of a class with its own copy of member variables and member functions that operate on these member variables.



A first basic example

Let’s create a class for a person that includes

- ☑ firstname (string)
- ☑ lastname (string)
- ☑ gender (int)

And constructs the full identity of the person (“Mr Dom Ginhac”).

Photo by Ben Sweet on Unsplash

Declaration in .h / Definition in .cpp

Header file

.h is #included in .cpp

Source file

Interface

Declaration of classes, functions, ...

Implementation

Definition of how it is implemented

USE SEPARATE FILES FOR THE DECLARATION AND THE DEFINITION OF CLASSES

So, when you define a new class Person, you write Person.h and Person.cpp and you include Person.h in each source file that uses Person objects.

02-Person/person.h

```
class Person {

public:
  Person(std::string firstname,
std::string lastname, int gender);
  std::string getFullName();

private:
  std::string _firstname;
  std::string _lastname;
  int _gender;
};
```

Declaration of a Class

Syntax

The keyword class and the name of the class introduce the declaration.

Brace brackets surround the contents.

Public declarations are generally made before private declarations.

Don't forget the final semi-colon!

Variables

Member variables are generally private.

Variables are prefixed by "_".

Functions

Functions are generally public.

getFullName() is only declared with input/output data types. Definition is elsewhere.

The special function named Person() is the constructor of the class that initializes the member variables when a new object is created.

A reminder on Functions

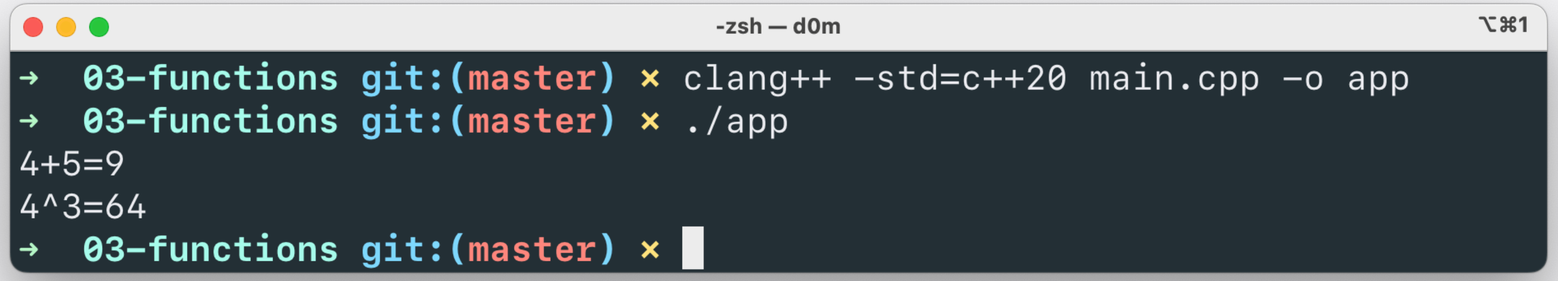
What is a function?

Block of statements that take specific input, does some computations, and finally produces a result as output.

Two types of functions: library functions and user-defined functions

03-functions/main.cpp

```cpp
int add(int x1, int x2) {
    return x1+x2;
}
int main() {
    int result = add(4,5);
    std::cout << "4+5=" << result << std::endl;
    int a = 4; int b=3;
    result = pow(a,b);
    std::cout << a << “^" << b << "=" << result << std::endl;
    return 0;
}
```



02-Person/person.cpp

```cpp
#include "person.h"

Person::Person(std::string f, std::string l, int g) {
  _firstname = f;
  _lastname = l;
  _gender = g;
}

std::string Person::getFullName() {
  std::string gender;
  if (_gender==1) {
    gender = "Mr";
  }
  else {
    gender = "Ms";
  }
  return gender + " " + _firstname + " " + _lastname;
}
```

Definition of a Class

Syntax

Do not forget to include “person.h” to get access to the declaration of the class.

Use the fully qualified name for the definition of each function.

Functions access member variables with no special syntax.

The constructor initializes the member variables with the values passed as parameters.

Other functions output data with the “return” instruction.

Using the class

02-Person/main.cpp

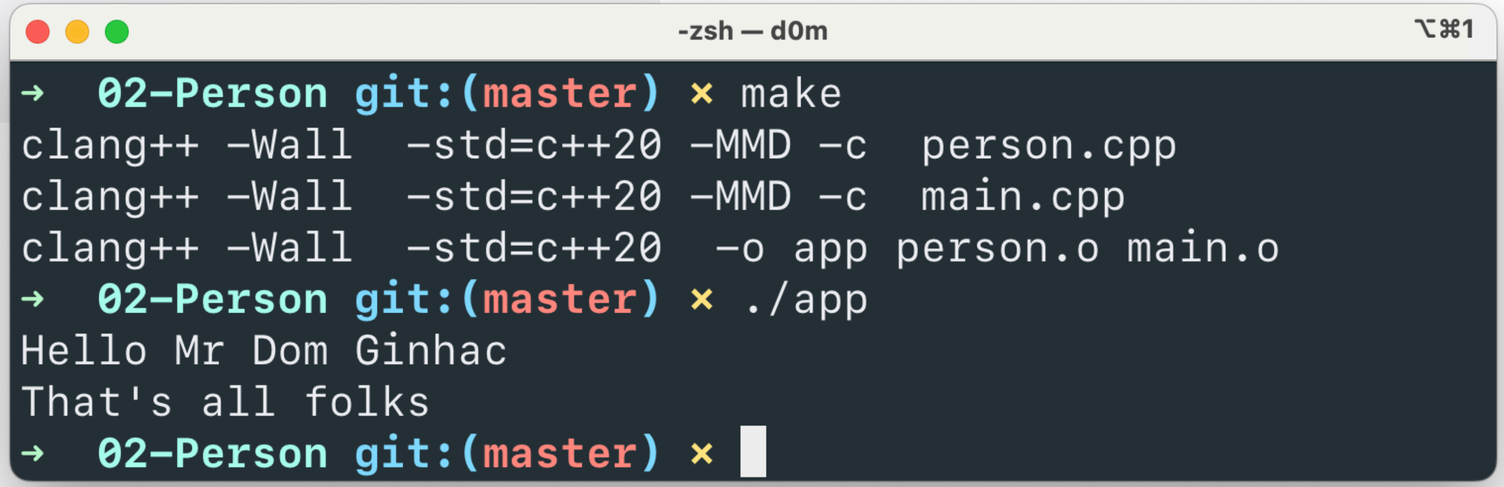
```cpp
#include <iostream>
#include "person.h"

int main(int argc, char const *argv[]) {
  Person p("Dom", "Ginhac", 1);
  std::string fullname = p.getFullName();
  std::cout << "Hello " << fullname << std::endl;
  std::cout << "That's all folks" << std::endl;
  return 0;
}
```

Syntax

Do not forget to include “person.h”.

Create a Person variable (p).

Access to class function with p.getFullName().



Questions

AGENDA

01 – Basics of Objects / Classes

02 – A realistic example of class

03 – More on functions

04 – Other user-defined types

05 – Organize your types in namespaces

A first realistic example of class

Suppose we **need** **dates** for an application.

What is the star wars day? May, 4 (i.e 05/04)
What is my birthday? May, 26 (i.e 05/26)

04-time/main.cpp

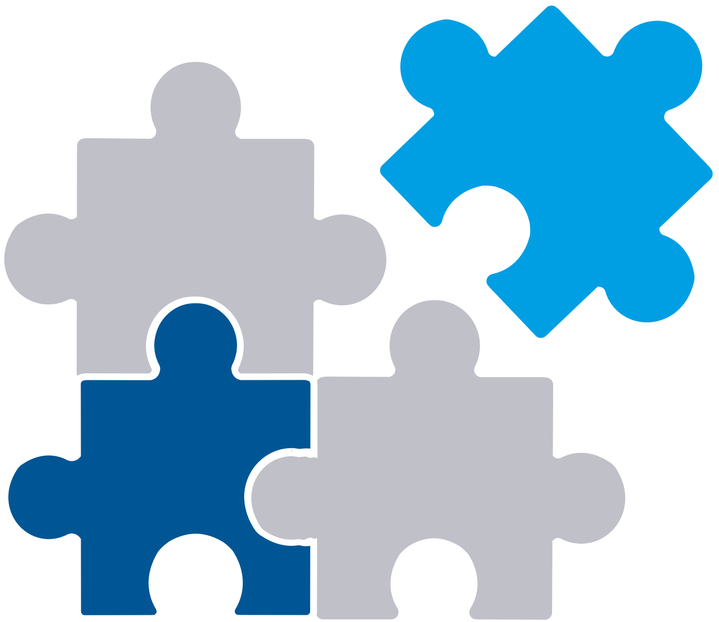
```cpp
#include <ctime>
#include <iostream>

int main(int argc, char const *argv[]) {
    std::time_t result = std::time(nullptr);
    std::cout << std::asctime(std::localtime(&result))
              << result << " seconds since the Epoch\n";
    return 0;
}
```

```
-zsh — d0m                                                    ⌥⌘1
→  04-time git:(master) ✗ clang++ -std=c++20 main.cpp -o app
→  04-time git:(master) ✗ ./app
Thu Sep 29 16:08:44 2022
1664460524 seconds since the Epoch
→  04-time git:(master) ✗
```

C++ does not provide Date objects, only time-related types in ctime library that are not easy to use.

The **only** solution is to **create** our own **Date class**.

The first two pillars of OOP



The design of a class must be made with respect to these 2 pillars.

Abstraction

First pillar of OOP, omnipresent in all programming concepts.

Mechanism of hiding the implementation details from the user, only the functionality will be provided to the user.

Abstraction expresses the intent of a class, i.e. the information on what the object does instead of how it does it.

Encapsulation

Mechanism, also known as Data hiding, that refers to the bundling of data/methods into a single coherent unit.

Restrict the direct access to some data/functions to prevent misuse and errors.

Member variables are kept private and methods are made public to control access to the data.

Date class declaration

Variables

We need two variables month and day represented as integers.

```
int _month; // Use snake_case for naming variables
int _day;   // Use “m_” or “_” prefix for variables
```

Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // functions with explicit names
```

Photo by Charles Deluvio on Unsplash

Date class declaration

Variables

We need two variables month and day represented as integers.

```
int _month; // Use snake_case for naming variables
int _day;   // Use “m_” or “_” prefix for variables
```

Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // functions with explicit names
```

Photo by Charles Deluvio on Unsplash

Variables

Public

Always available

- Public variables form an interface to the class and are accessible outside the class.
- Imagine a Date class in which everyone can code a month outside the range [1-12].
- Imagine a Bank account class in which everyone can change the owner of the account or the balance.

```
class Date {
public:
    int _month;
    int _day;
};
```

VS

```
class Date {
private:
    int _month;
    int _day;
};
```

- Private variables are not accessible outside the class; they can be accessed only through methods of the class.
- Need to write public functions to initialize, read, or write variables.
- Encapsulation protects the objects by hiding their internal representation from the outside.

Encapsulated variables

Private

Always hide the internal representation of a class with

Private

Variables

Date class declaration

Variables

We need two variables month and day represented as integers.

```
int _month; // Use snake_case for naming variables
int _day;   // Use “m_” or “_” prefix for variables
```

Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // functions with explicit names
```

Photo by Charles Deluvio on Unsplash

C++ Class Constructor

Special class functions

Constructors allocate storage to the new created objects and perform initialization of each new object.

How to create a constructor?

Constructors have the same name as the class itself and can take arguments that are used to initialize the member variables.

A class can have several constructors with different parameters.

No return type for constructors.

How to use a constructor?

Constructors are automatically called when a new object is created. No need to explicitly call them.

Photo by Silvia Brazzoduro on Unsplash

3 TYPES OF CONSTRUCTORS

Minimal

The most basic constructor with no input parameter.

Date d;

Parametrized

It's a customized constructor with parameters.

Date pi_day(3,14);

Copy

Used to declare and initialise an object from another object.

Date another_pi_day = pi_day;

NO

Constructor

For each user-defined class, C++ provides a default implicit constructor that is eventually called when an object is created. Be careful, in our Date class, the int variables are automatically initialized to zero or to random values, leading to an invalid date.

05-Date-no-constructor/main.cpp

```cpp
#include "date.h"
int main(int argc, char const *argv[]) {
    Date d1;
    std::cout << "d1: " << d1.month() << "/" << d1.day();
    Date d2 = Date();
    std::cout << " d2: " << d2.month() << "/" << d2.day();
    Date d3{};
    std::cout << " d3: " << d3.month() << "/" << d3.day()
              << std::endl;
    return 0;
}
```

Best practices

Always provide an explicit constructor that creates valid objects.

MINIMAL

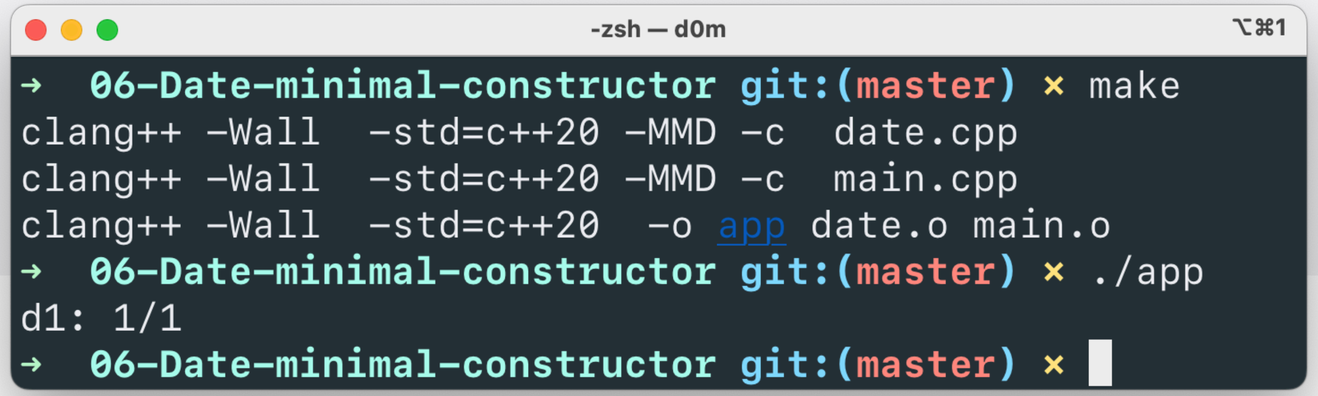
Constructor

06-Date-minimal-constructor/date.h

```
class Date {
public:
    Date();
    int month();
    int day();
private:
    int _month;
    int _day;
};
```

06-Date-minimal-constructor/date.cpp

```
Date::Date() {
    _month = 1;
    _day = 1;
}
```



A minimal constructor initializes each new object with default values hardcoded in the constructor.

PARAMETRIZED

Constructor

07-Date-parametrized-constructor/date.h

```
class Date {
public:
    Date();
    Date(int month, int day);
    int month();
    int day();
private:
    int _month;
    int _day;
};
```

07-Date-parametrized-constructor/date.cpp

```
Date::Date() {
    _month = 1;
    _day = 1;
}
```

```
Date::Date(int month, int day) {
    _month = month;
    _day = day;
}
```

A customized constructor with parameters can be added to the class.

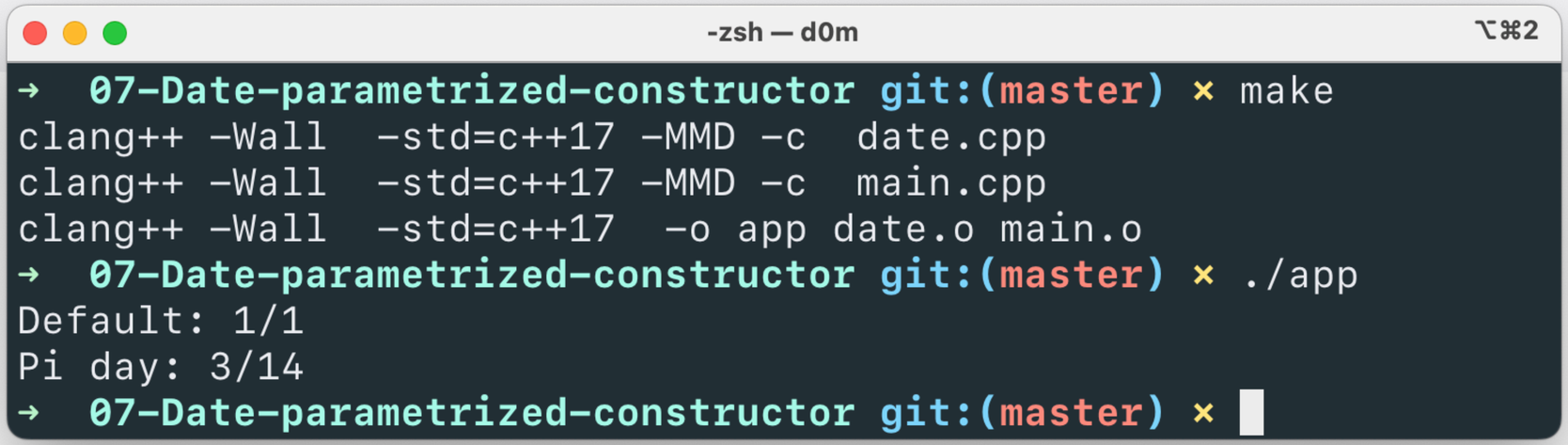
Each new object is initialized with specific values passed as arguments of the constructor.

PARAMETRIZED

Constructor

07-Date-parametrized-constructor/main.cpp

```cpp
#include "date.h"
int main(int argc, char const *argv[]) {
  Date d;
  std::cout << "Default: " << d.month() << "/" << d.day() << std::endl;
  Date pi_day(3,14);
  std::cout << "Pi day: " << pi_day.month() << "/" << pi_day.day() << std::endl;
  return 0;
}
```



PARAMETRIZED

Constructor

If a class is defined only with a parametrized constructor, the compiler will not generate a default constructor.

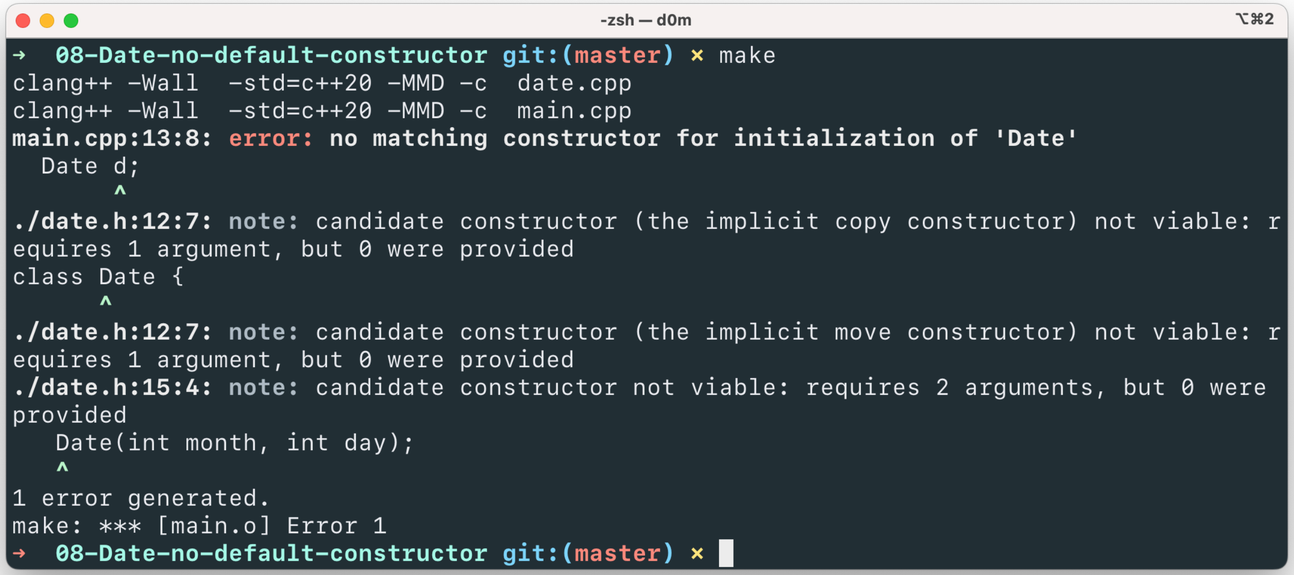
If we create a new object:

```
Date d;
```

The compiler will complain that we have no default constructor.

We can force the compiler to automatically provide this constructor by using the default specifier (C++11 extension).

Here, the default constructor randomly initializes the variables, leading to invalid dates.




```
class Date {
public:
    Date() = default;
    Date(int month, int day);
    int month();
    int day();
```

Reducing the #constructors

Constructors code is often somewhat redundant.

Default and parametrized constructors are overloaded functions (i.e. the same name, but different and unique parameters !)

Using default values expresses that there is really just ONE constructor to provide.

09-Date-reducing-constructors/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
private:
    int _month;
    int _day;
};
```

09-Date-reducing-constructors/date.cpp

```
Date::Date(int month, int day) {
    _month = month;
    _day = day;
}
```

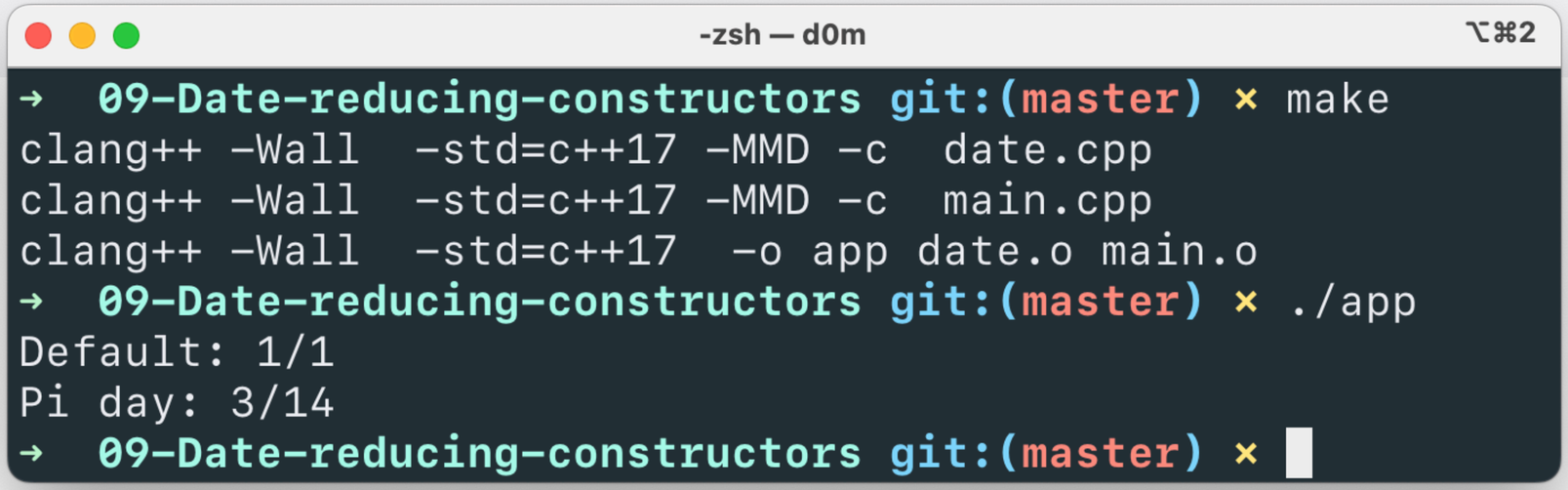
Default values are explicitly indicated in the declaration of the constructor (.h).

The definition of the constructor (.cpp) remains unchanged.

Reducing the #constructors

09-Date-reducing-constructors/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  Date d;
  std::cout << "Default: " << d.month() << "/" << d.day() << std::endl;
  Date pi_day(3,14);
  std::cout << "Pi day: " << pi_day.month() << "/" << pi_day.day() << std::endl;
  return 0;
}
```



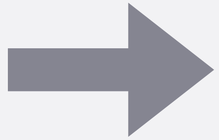
Member Initialiser lists

Our class member data are initialized using the assignment operator. It works perfectly for most of the types but does not work in some specific cases including const, references, ...

C++ provides “Member initialiser list” starting with a colon after the list of parameters.

10-Date-Initializer/date.cpp

```
Date::Date(int month, int day) {
    _month = month;
    _day = day;
}
```



10-Date-Initializer/date.cpp

```
Date::Date(int month, int day) :
_month(month), _day(day) {
// Nothing to do
}
```

Can be used with any user-defined type, more concise and more efficient.

Best practices

Preferably use Initialiser lists rather than assignments.

COPY

Constructor

Photo by Sharon McCutcheon on Unsplash

Used to declare and initialise an object from another object in the following 3 cases:

Object constructed from another object

```
Date starwars(5,4);
Date other_starwars_day = starwars;
Date another_starwars(starwars);
```

Object returned by a function

```
Date tomorrow = today.nextDay();
```

Object passed to a function

```
Date pi_day(3,14);
bool b = starwars.before(pi_day);
```

COPY

Constructor

Do we need writing specific code for Copy constructor?

NO because the C++ compiler creates a default copy constructor making a memberwise copy.

11-Date-copy-constructor/main.cpp

```cpp
#include "date.h"
#include <iostream>

int main(int argc, char const *argv[]) {
  Date starwars(5,4);
  std::cout << "1: " << starwars.month() << "/" << starwars.day() << std::endl;
  Date s2 = starwars;
  std::cout << "2: " << s2.month() << "/" << s2.day() << std::endl;
  Date s3(starwars);
  std::cout << "3: " << s3.month() << "/" << s3.day() << std::endl;
  return 0;
}
```

How to create a valid Date object

?

Photo by Sigmund on Unsplash

Check a date

12-Date-create-valid-object/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
private:
    int _month;
    int _day;
    bool isDate(int month, int day);
};
```

isDate is private because the function does not need to be reachable from outside.

isDate is used at the creation of a new Date.

12-Date-create-valid-object/date.cpp

```
bool Date::isDate(int month, int day) {
    if ((day < 1) || (day>31)) return false;
    if ((month <1) || (month>12)) return false;
    if ((month == 2) && (day > 28)) return false;
    if (((month == 4) || (month == 6) ||
         (month == 9) || (month == 11)) && (day > 30)) return false;
    return true;
}
Date::Date(int month, int day) : _month(month), _day(day) {
    bool status = isDate(month, day);
    assert(status && "Date is not valid");
}
```

|| = OR
&& = AND

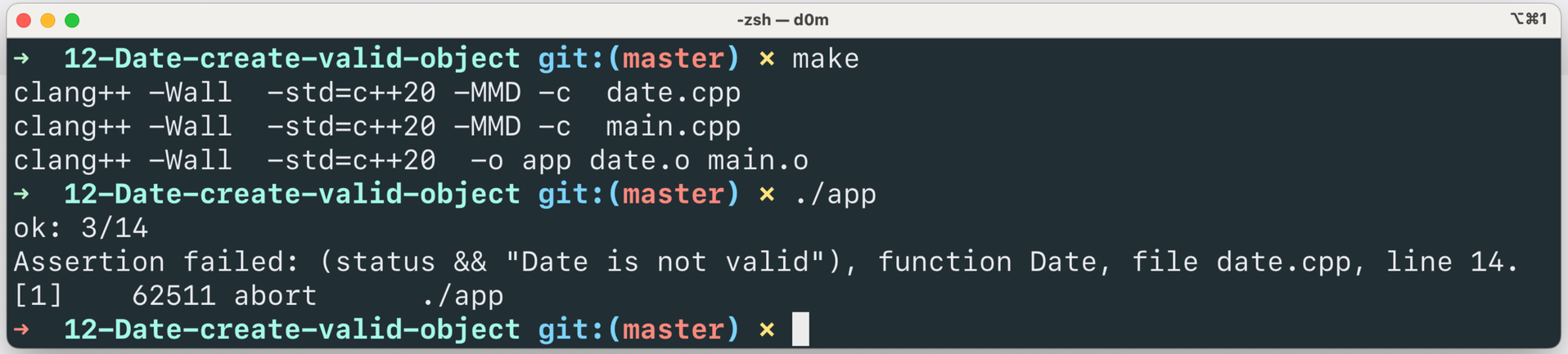
Using assertions to exit.

Lecture to come soon

Check a date at the creation of an object

12-Date-create-valid-object/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  Date pi_day_ok(3,14);
  std::cout << "ok: " << pi_day_ok.month() << "/"
            << pi_day_ok.day() << std::endl;
  Date pi_day_error(14,3);
  std::cout << "nok: " << pi_day_error.month() << "/"
            << pi_day_error.day() << std::endl;
  return 0;
}
```



RULES #02

Constructor

Always design user-defined types so that new objects are guaranteed to be valid.

Provide constructors that create only valid objects.
Use Member initialiser list for initializing variables.

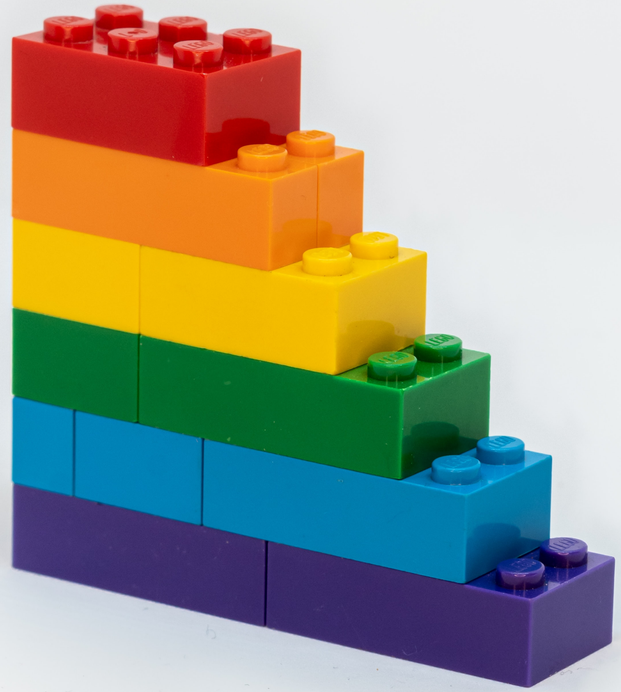


Photo by Xavi Cabrera on Unsplash

Date class declaration

Variables

We need two variables month and day represented as integers.

```
int _month; // Use snake_case for naming variables
int _day;   // Use “m_” or “_” prefix for variables
```

Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // functions with explicit names
```

Photo by Charles Deluvio on Unsplash

GETTERS

Methods

13-Date-getters/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
private:
    int _month;
    int _day;
};
```

13-Date-getters/date.cpp

```
Date::Date(int month, int day) :
_month(month), _day(day) {
}
int Date::month()  {
    return _month;
}
int Date::day()  {
    return _day;
}
```

For each private variable that has to be read, a public get method must be defined.

Name them with Obj-C/Swift style (int Date::month()) or java style (int Date::getMonth()).

No input argument and return type is the same as the type of the member variable.

GETTERS

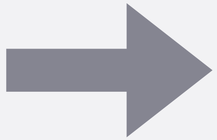
Const Methods

13-Date-getters/date.h

```
int month();
int day();
```

13-Date-getters/date.cpp

```
int Date::month()  {
    return _month;
}
int Date::day()  {
    return _day;
}
```

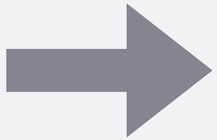


14-Date-const-getters/date.h

```
int month() const;
int day() const;
```

14-Date-const-getters/date.cpp

```
int Date::month() const {
    return _month;
}
int Date::day() const {
    return _day;
}
```



"const method" informs compiler that you will not modify the object on which this method is called. So if you try to modify your object inside the method, then compiler will issue error.

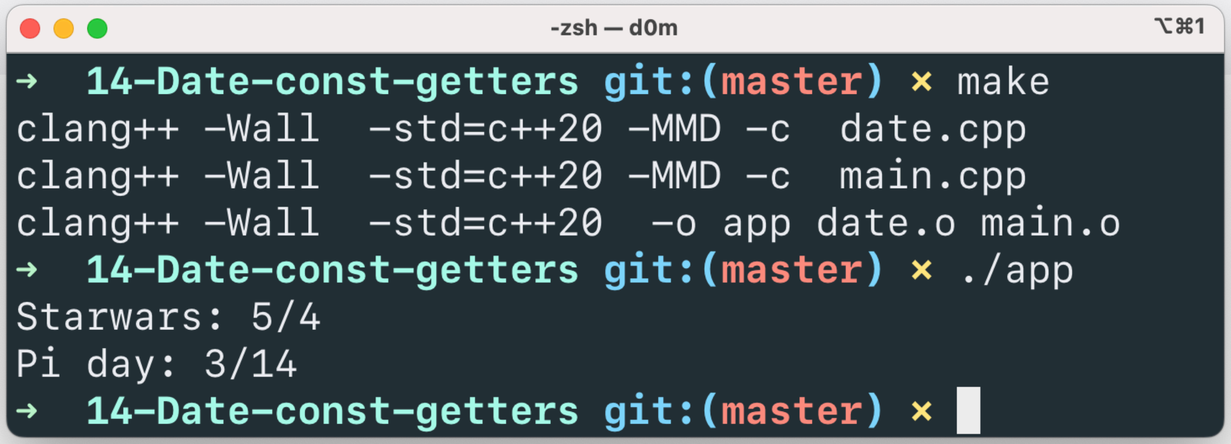
Always mark functions as const unless you can't. Getters should always be marked as const because they only read the member variables.

Using const Getters

14-Date-const-getters/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  Date starwars(5,4);
  std::cout << "Starwars: " << starwars.month() << "/"
            << starwars.day() << std::endl;
  Date pi_day(3,14);
  std::cout << "Pi day: " << pi_day.month() << "/"
            << pi_day.day() << std::endl;
  return 0;
}
```

Using const getters is a very useful information for the compiler which can do optimizations and for humans who read and understand your code.



SETTERS

Methods

15-Date-setters/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
    void updateMonth(int month);
    void updateDay(int day);
…
};
```

15-Date-setters/date.cpp

```
void Date::updateMonth(int month) {
    _month = month;
}

void Date::updateDay(int day) {
    _day = day;
}
```

If a member variable needs to be modified after its initialization, a setter must be defined.

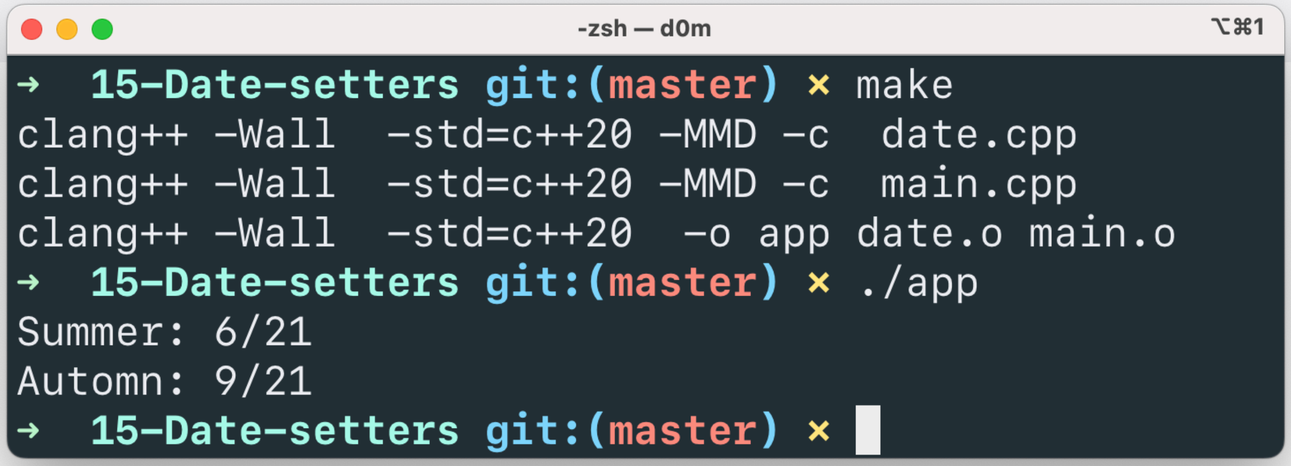
A setter can not be a const method because it updates the object.

Input argument has same type than member variable, no return type.

Using Setters

15-Date-setters/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  Date a_day(6,21);
  std::cout << "Summer: " << a_day.month() << "/" << a_day.day() << std::endl;
  a_day.updateMonth(9);
  std::cout << "Automn: " << a_day.month() << "/" << a_day.day() << std::endl;
  return 0;
}
```



Check a date when updating an object

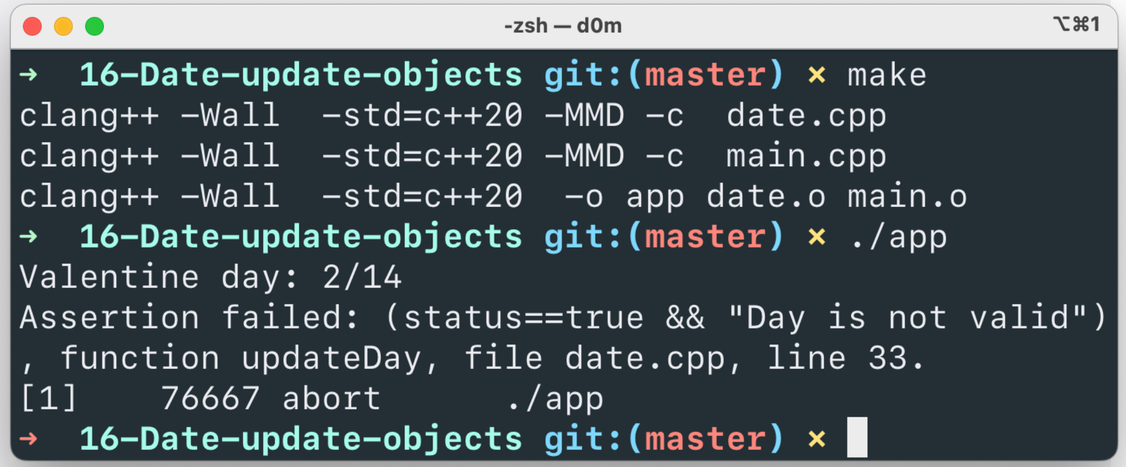
16-date-update-objects/date.cpp

```
void Date::updateMonth(int month) {
    bool status = isDate(month, _day);
    assert(status && "Month is not valid");
    _month = month;
}
```

```
void Date::updateDay(int day) {
    bool status = isDate(_month, day);
    assert(status && "Day is not valid");
    _day = day;
}
```

16-date-update-objects/main.cpp

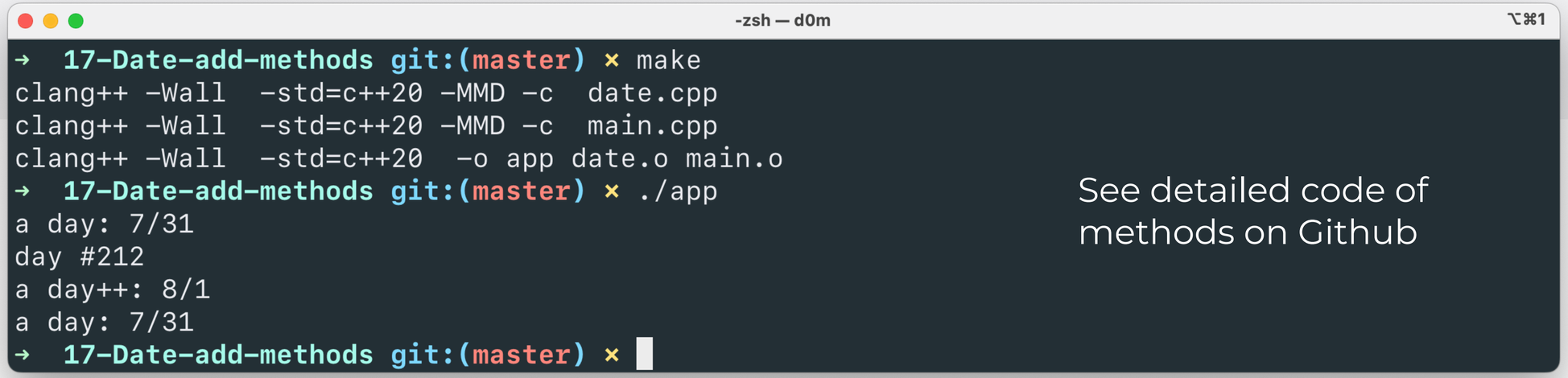
```
int main(int argc, char const *argv[]) {
  Date love(2,14);
  std::cout << "Valentine day: " << love.month()
            << "/" << love.day() << std::endl;
  love.updateDay(30);
  std::cout << “NOK 30/02: " << love.month()
            << "/" << love.day() << std::endl;
  return 0;
}
```



Adding some methods to the class

17-Date-add-methods/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  Date a_day(7,31);
  std::cout << "a day: " << a_day.month() << "/" << a_day.day() << std::endl;
  std::cout << "day #" << a_day.dayOfYear() << std::endl;
  a_day.next();
  std::cout << "a day++: " << a_day.month() << "/" << a_day.day() << std::endl;
  a_day.back();
  std::cout << "a day: " << a_day.month() << "/" << a_day.day() << std::endl;
  return 0;
}
```



See detailed code of methods on Github

Photo by Egor Myznik on Unsplash

#03 How to choose between Getters, Setters or Methods

1. Some variables may be entirely internal to the object, and should have neither getters nor setters.
2. Some variables should be read-only, so they may need getters but not setters.
3. Some variables need to be in read-write mode, and should have both getters and setters.
4. Some variables may need to be kept consistent with each other. Do not provide a setter for each one, but rather a single method for setting them at the same time, so that you can check the values for consistency.
5. Some variables may only need to be changed in a certain way, such as incremented or decremented. You should provide adequate methods, rather than setters.

Function

A function is a block of statements that take specific inputs, does some computations, and finally produces a result as output.

A function is defined independently of any other code and can be used anywhere in the program.

Method

A method also works as a function.

A method is declared / defined into a class, belongs to an object of this class, and only be invoked by its object.

A method is able to operate on data that is contained within the object.

FUNCTION

vs

METHOD

METHOD
VS
FUNCTION

Method

```
int Date::dayOfYear() const {
    auto day=0;
    for (auto i=1;i<_month;i++) {
        day+=getDaysInMonth(i);
    }
    day+= _day;
    return day;
}
```

Function

```
int dayOfYear(Date d) {
    auto day=0;
    for (auto i=1;i<d.month();i++) {
        day+=getDaysInMonth(i);
    }
    day+= d.day();
    return day;
}
```

See detailed code of getDaysInMonth() on Github

Method vs Function

18-Date-method-vs-function/main.cpp

```
nt main(int argc, char const *argv[]) {
  Date a_day(7,31);
  std::cout << "a day: " << a_day.month() << "/" << a_day.day() << std::endl;
  std::cout << "helper function -> day #" << dayOfYear(a_day) << std::endl;
  std::cout << "method -> day #" << a_day.dayOfYear() << std::endl;
  return 0;
}
```

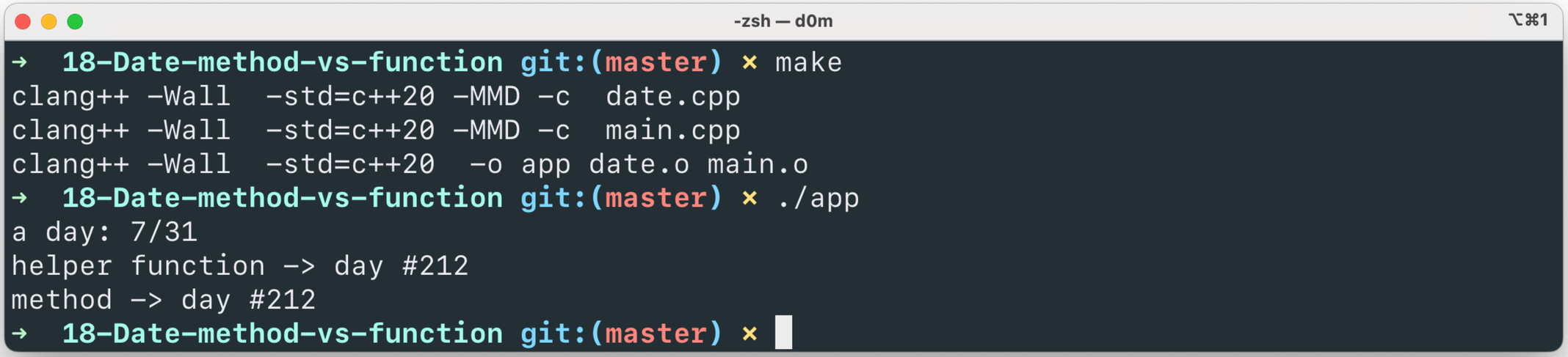


Photo by Jon Flobrant on Unsplash

RULES #04

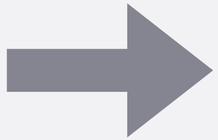
How to choose between Methods and Functions

1. Design classes to be minimal to avoid complex code modification when a change to the representation is considered.
2. Make a function a method only if it needs direct access to the representation of a class, i.e. access to the private variables.
3. Every other function must be defined as a “helper function”, i.e. a function that can be associated with the class but does not require a direct access to the internal representation of the class.
4. Declare helper functions in the .h file of the class and define helper functions in the .cpp file of the class.

Final Date class

18-Date-method-vs-function/date.h

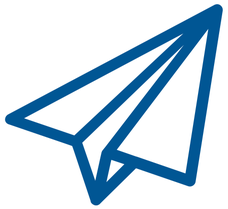
```
class Date {
public:
    Date(int month=1, int day=1);
    int month() const;
    int day() const;
    void updateMonth(int month);
    void updateDay(int day);
    int dayOfYear() const;
    std::string toString() const;
    void next();
    void back();
private:
    int _month;
    int _day;
    bool isDate(int month, int day) const;
    int getDaysInMonth(int month) const;
};
```



19-Date-final-class/date.h

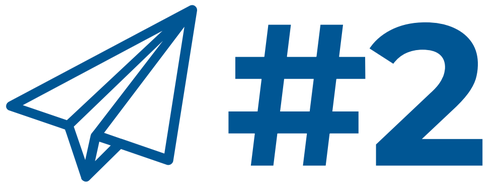
```
class Date {
public:
    Date(int month=1, int day=1);
    int month() const;
    int day() const;
    void updateMonth(int month);
    void updateDay(int day);
    void next();
    void back();
private:
    int _month;
    int _day;
};
int dayOfYear(Date d);
std::string toString(Date d);
bool isDate(int month, int day);
int getDaysInMonth(int month);
```

A SECOND TAKE HOME MESSAGE ABOUT CLASSES



#2

1. Avoid large classes. Adopt Single Responsibility Principle and design classes with few data/methods are better.
2. Add as many helper functions as you want (Open Closed Principle, i.e Open for extension, but closed for modification).
3. Promote Abstraction and Encapsulation by design easy-to-use classes that hide all the implementation détails.
4. Always create / update valid objects.



#2

DRY, KISS & YAGNI Principles

Don’t Repeat Yourself

You should avoid duplication of code.

Do not use copy/paste.

Keep It Simple Stupid

You should avoid unnecessary complexity.

Make your code simple

You Ain’t Gonna Need It

You should not create code that is not really necessary.

Remove unnecessary functionality and logic.

Questions

AGENDA

01 – Basics of Objects / Classes

02 – A realistic example of class

03 – More on functions

04 – Other user-defined types

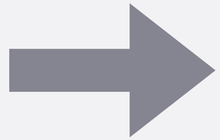
05 – Organize your types in namespaces

Less methods?

Can we refactor the next() and back() methods into functions ?

19-Date-final-class/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
    int month() const;
    int day() const;
    void updateMonth(int month);
    void updateDay(int day);
    void next();
    void back();
private:
    int _month;
    int _day;
};
```



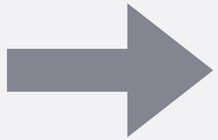
20-Date-less-methods/date.h

```
class Date {
public:
    Date(int month=1, int day=1);
    int month() const;
    int day() const;
    void updateMonth(int month);
    void updateDay(int day);
private:
    int _month;
    int _day;
};
void next(Date d);
void back(Date d);
```

Deep study of next()

19-Date-final-class/date.cpp

```
void Date::next() {
  if ((_month==12) && (_day==31)) {
      _day=1;
      _month=1;
  }
  else if (_day==getDaysInMonth(_month)) {
      _day=1;
      _month++;
  }
  else {
      _day++;
  }
}
```



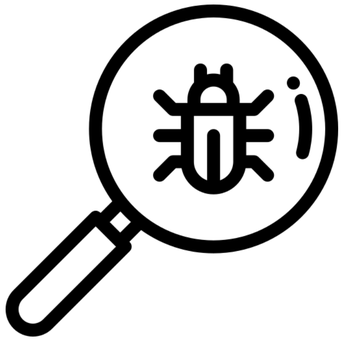
20-Date-less-methods/date.cpp

```
void next(Date d) {
  if ((d.month()==12) && (d.day()==31)) {
      d.updateDay(1);
      d.updateMonth(1);
  }
  else if (d.day()==getDaysInMonth(d.month()
      d.updateDay(1);
      d.updateMonth(d.month()+1);
  }
  else {
      d.updateDay(d.day()+1);
  }
}
```

Less methods?

20-Date-less-methods/main.cpp

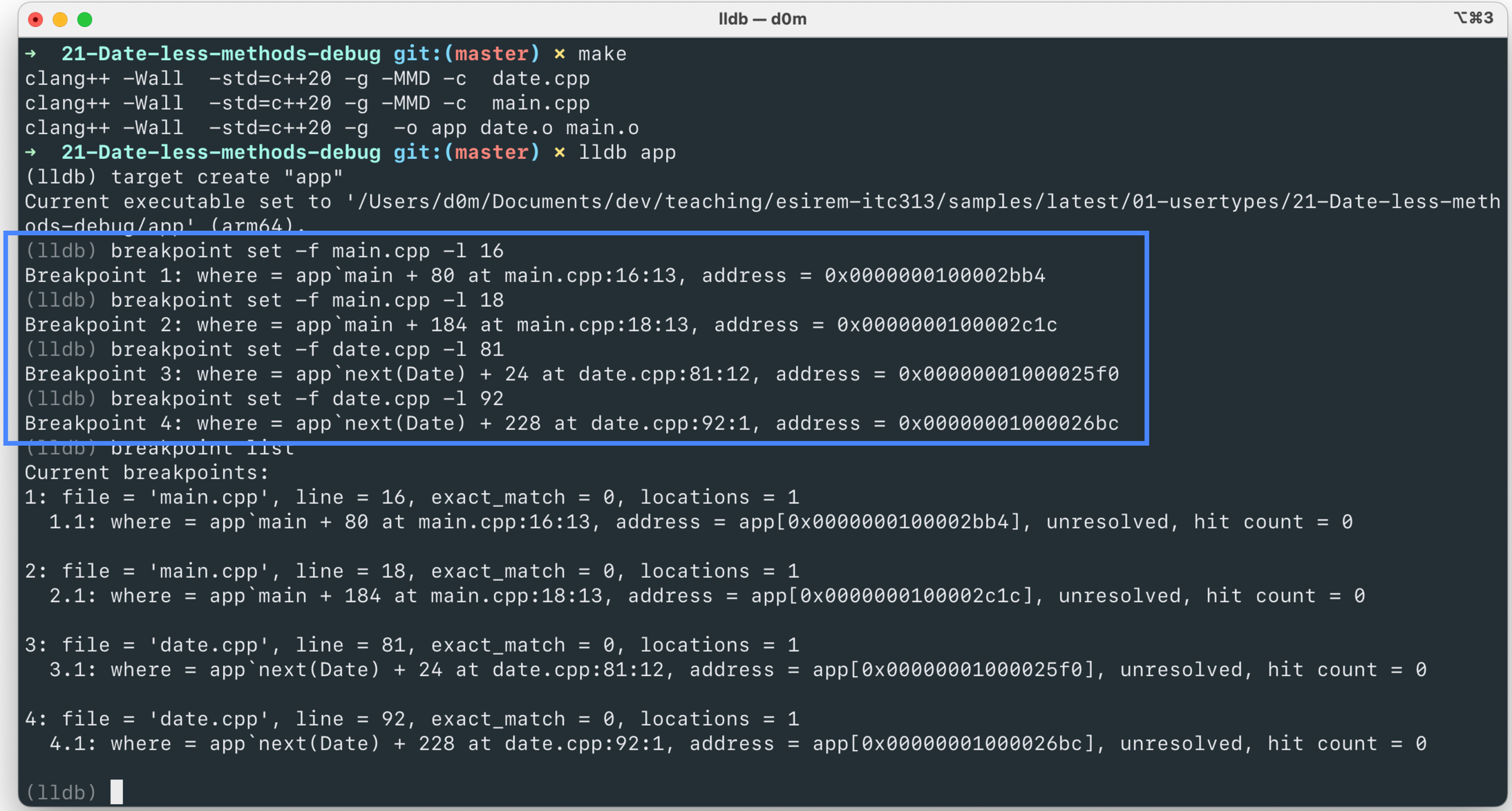
```cpp
int main(int argc, char const *argv[]) {
  Date a_day(7,31);
  std::cout << "a day: " << toString(a_day) << std::endl;
  next(a_day);
  std::cout << "a day + 1: " << toString(a_day) << std::endl;
  back(a_day);
  std::cout << "a day - 1: " << toString(a_day) << std::endl;
  return 0;
}
```



```
-zsh — d0m                                                    ⌥⌘1
→  20-Date-less-methods git:(master) ✗ make
clang++ -Wall  -std=c++20 -MMD -c  date.cpp
clang++ -Wall  -std=c++20 -MMD -c  main.cpp
clang++ -Wall  -std=c++20  -o app date.o main.o
→  20-Date-less-methods git:(master) ✗ ./app
a day: 7/31
a day + 1: 7/31
a day - 1: 7/31
→  20-Date-less-methods git:(master) ✗
```

Time to DEBUG!

Time to debug



Time to debug

```
lldb — d0m                                                    ⌥⌘3
(lldb) run
There is a running process, kill it and restart?: [Y/n] y
Process 72495 exited with status = 9 (0x00000009)
Process 72506 launched: '/Users/d0m/Documents/dev/teaching/esirem-itc313/samples/latest/01-usertypes/21-Date-less-meth
ods-debug/app' (arm64)
Process 72506 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 4.1
    frame #0: 0x0000000100002bb4 app`main(argc=1, argv=0x000000016fdff5b8) at main.cpp:16:13
   13
   14   int main(int argc, char const *argv[]) {
   15     Date a_day(7,31);
-> 16     std::cout << "a day: " << toString(a_day) << std::endl;
   17     next(a_day);
   18     std::cout << "a day + 1: " << toString(a_day) << std::endl;
   19     back(a_day);
Target 0: (app) stopped.
(lldb) frame variable -L a_day
0x000000016fdff418: (Date) a_day = {
0x000000016fdff418:   _month = 7
0x000000016fdff41c:   _day = 31
}
(lldb) continue
Process 72506 resuming
a day: 7/31
Process 72506 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 3.1
    frame #0: 0x00000001000025f0 app`next(d=(_month = 7, _day = 31)) at date.cpp:81:12
   78
   79
   80   void next(Date d) {
-> 81       if ((d.month()==12) && (d.day()==31)) {
   82           d.updateDay(1);
   83           d.updateMonth(1);
   84       }
Target 0: (app) stopped.
(lldb) frame variable -L d
0x000000016fdff328: (Date) d = {
0x000000016fdff328:   _month = 7
0x000000016fdff32c:   _day = 31
}
(lldb)
```

Time to debug

```
lldb — d0m                                                    ⌥⌘3
(lldb) continue
Process 72506 resuming
Process 72506 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x00000001000026bc app`next(d=(_month = 8, _day = 1)) at date.cpp:92:1
   89       else {
   90           d.updateDay(d.day()+1);
   91       }
-> 92   }
   93
   94   void back(Date d) {
   95       if ((d.month()==1) && (d.day()==1)) {
Target 0: (app) stopped.
(lldb) frame variable -L d
0x000000016fdff328: (Date) d = {
0x000000016fdff328:   _month = 8
0x000000016fdff32c:   _day = 1
}
(lldb) continue
Process 72506 resuming
Process 72506 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100002c1c app`main(argc=1, argv=0x000000016fdff5b8) at main.cpp:18:13
   15     Date a_day(7,31);
   16     std::cout << "a day: " << toString(a_day) << std::endl;
   17     next(a_day);
-> 18     std::cout << "a day + 1: " << toString(a_day) << std::endl;
   19     back(a_day);
   20     std::cout << "a day - 1: " << toString(a_day) << std::endl;
   21
Target 0: (app) stopped.
(lldb) frame variable -L a_day
0x000000016fdff418: (Date) a_day = {
0x000000016fdff418:   _month = 7
0x000000016fdff41c:   _day = 31
}
(lldb)
```

Functions and arguments

21-Date-less-methods-debug/main.cpp

```
int main(int argc, char const *argv[]) {
  Date a_day(7,31);
  std::cout << "a day: " << toString(a_day) << std::endl;
  next(a_day);
  std::cout << "a day + 1: " << toString(a_day) << std::endl;
  return 0;
}
```

What happened?

The next function takes a Date object as input parameter.
But this Date object is not the a_day variable but a copy at a different place in memory.
The copy is modified into the function but the original variable is never altered by the code inside the function.

This mechanism called “**Pass by Values**” is the default method of passing arguments to a function.

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values

Default behavior

Pass by References

Specific to C++

Pass by Pointers

Old school C-based mechanism

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values

Default behavior

Pass by References

Specific to C++

Pass by Pointers

Old school C-based mechanism

What's a reference?

A reference is a type of C++ variable that acts as an alias to another object or value.

```
int value = 5;          // value is 5
int& number = value; // reference to value
Date love(2,14);       // Valentine's day
Date& day = love;     // reference to valentine's day
```

A reference acts identically to the value it's referencing. You can access to the value with the variable or the reference.

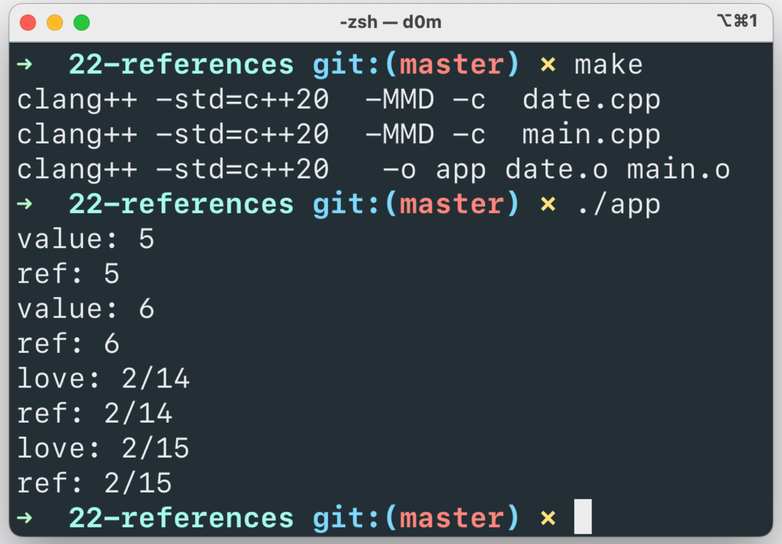
A reference must always be initialized when created.

Once initialized, it can't be changed to reference another variable.

What's a reference?

22-references/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  int value = 5; // value is 5
  int& number = value; // reference to variable value
  std::cout << "value: " << value << std::endl;
  std::cout << "ref: " << number << std::endl;
  value = 6;
  std::cout << "value: " << value << std::endl;
  std::cout << "ref: " << number << std::endl;
  Date love(2,14);      // Valentine's day
  Date& day = love;     // reference to valentine's day
  std::cout << "love: " << toString(love) << std::endl;
  std::cout << "ref: " << toString(love) << std::endl;
  day.next();
  std::cout << "love: " << toString(love) << std::endl;
  std::cout << "ref: " << toString(love) << std::endl;
  return 0;
}
```



References and functions

References are most often used as Function parameters.

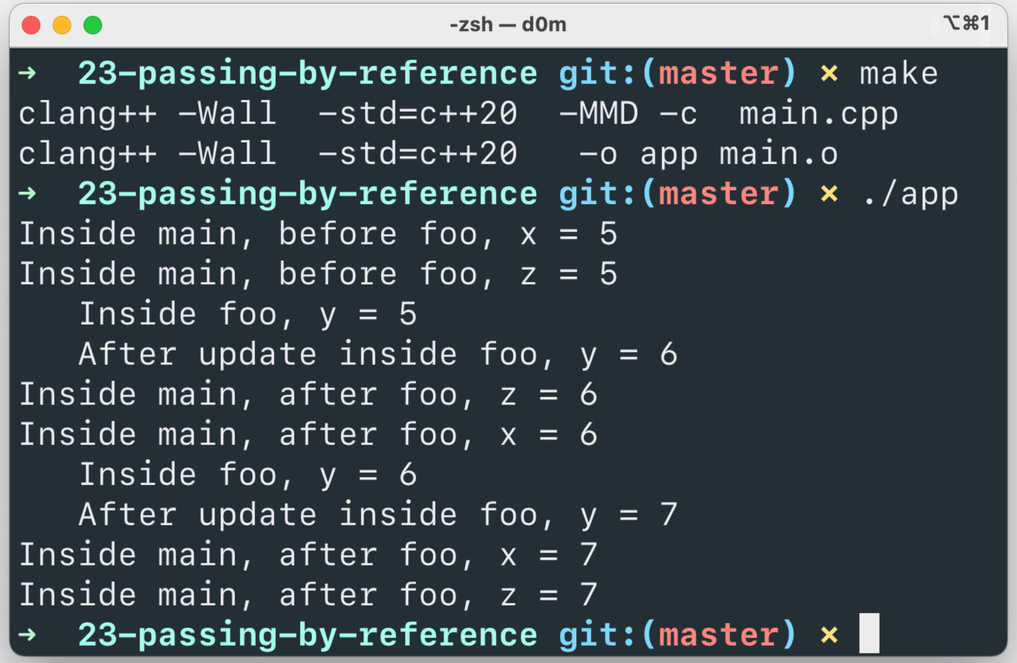
The reference parameter acts as an alias for the argument, and no copy of the argument is made.

A function that uses a reference parameter is able to modify the argument passed in.

```
void foo(int& y) {
    std::cout << "Inside foo, y = " << y << std::endl;
    y++;
    std::cout << "After update inside foo, y = " << y << std::endl;
}
```

23-passing-by-reference/main.cpp

```cpp
void foo(int& y) {
    std::cout << "   Inside foo, y = " << y << std::endl;
    y++;
    std::cout << "   After update inside foo, y = " << y << std::endl;
}
int main(int argc, char const *argv[]) {
    int x = 5;
    int& z = x; // create a reference
    std::cout << "Inside main, before foo, x = " << x << std::endl;
    std::cout << "Inside main, before foo, z = " << z << std::endl;
    foo(z); // Call foo with the reference z
    std::cout << "Inside main, after foo, z = "
              << z << std::endl;
    std::cout << "Inside main, after foo, x = "
              << x << std::endl;
    foo(x); // Compiler automatically creates a ref
    std::cout << "Inside main, after foo, x = "
              << x << std::endl;
    std::cout << "Inside main, after foo, z = "
              << z << std::endl;
    return 0;
}
```



The next() function

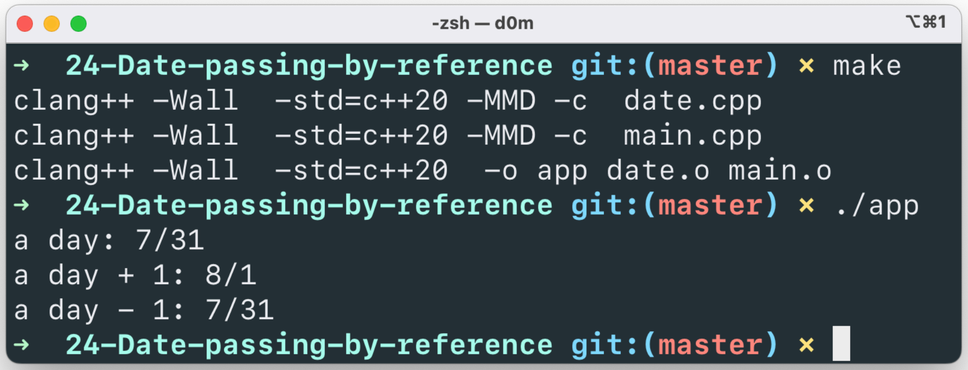
24-Date-passing-by-reference/date.h

```
void next(Date& d);
```

24-Date-passing-by-reference/date.cpp

```
void next(Date& d) {
    if ((d.month()==12) && (d.day()==31)) {
        d.updateDay(1);
        d.updateMonth(1);
    }
    else if (d.day()==getDaysInMonth(d.month())) {
        d.updateDay(1);
        d.updateMonth(d.month()+1);
    }
    else {
        d.updateDay(d.day()+1);
    }
}
```

Use a reference each time we call a Date object as an argument of a function, i.e for the next() and back() functions



It works now with the same code for main.cpp

References

PROs

- References allow a function to change the value of an argument
- References can be used to return multiple values from a function
- “Pass by reference” is fast

CONs

- It can be hard to tell whether an argument passed as a reference is an input, an output or both
- In the body of a function, it’s not possible to make the difference between a value and a reference

#05

1. Use “Pass by reference” when there is a need to modify data or when passing complex objects.
2. Use “Pass by value” when passing fundamental types that don’t need to be modified.

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values

Default behavior

Pass by References

Specific to C++

Pass by Pointers

Old school C-based mechanism

Questions

The concept of pointers

Pointers are one of the most distinct and exciting features of C/C++ languages.

Pointers are used to access and manipulate the memory using addresses.

Pointers are variables that store adresses.

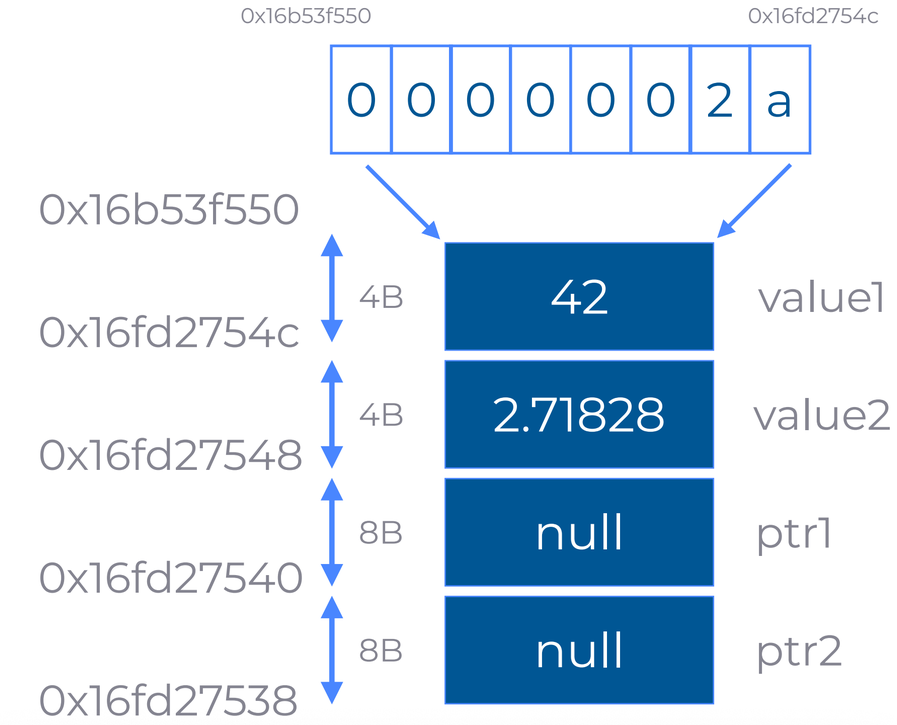
Pointers look like references but they are not references.

Photo by Nathalie SPEHNER on Unsplash

Declaring pointers (type*)

25-declaring-pointers/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  // Declare and Init some variables
  int value1 = 42;
  float value2 = 2.71828;
  int* ptr1 = nullptr; // int *ptr = nullptr;
  float* ptr2 = nullptr;

  // prints the memory address of the variables
  std::cout << "@value1: " << &value1 << std::endl;
  std::cout << "@value2: " << &value2 << std::endl;
  std::cout << "@ptr1:   " << &ptr1 << std::endl;
  std::cout << "@ptr2:   " << &ptr2 << std::endl;
  return 0;
}
```




```
-zsh — d0m                                    ⌥⌘1
→  25-declaring-pointers git:(master) ✗ make
clang++ -std=c++20 -MMD -c  main.cpp
clang++ -std=c++20  -o app main.o
→  25-declaring-pointers git:(master) ✗ ./app
@value1: 0x16fd2754c
@value2: 0x16fd27548
@ptr1:   0x16fd27540
@ptr2:   0x16fd27538
→  25-declaring-pointers git:(master) ✗
```

Assigning pointers (& operator)

26-assigning-pointers/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  int value1 = 42;
  int* ptr1 = nullptr;
  // Makes ptr point to the integer variable
  // using the address-of operator (&)
  ptr1 = &value1;

  // prints the memory address
  // and the contents of the variables
  std::cout << "@value1: " << &value1 << std::endl;
  std::cout << "@ptr1:   " << &ptr1 << std::endl;
  std::cout << "value1: " << value1 << std::endl;
  std::cout << "ptr1:   " << ptr1 << std::endl;
  return 0;
}
```

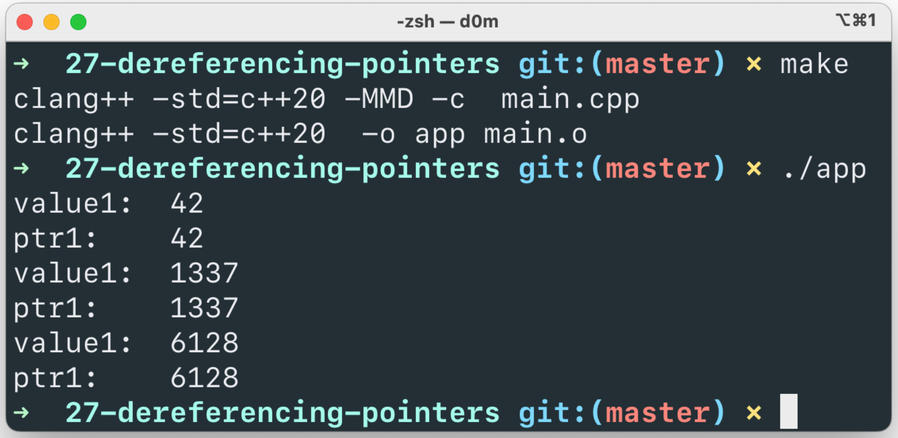

```
-zsh — d0m                                    ⌥⌘1
→  26-assigning-pointers git:(master) × make
clang++ -std=c++20 -MMD -c  main.cpp
clang++ -std=c++20  -o app main.o
→  26-assigning-pointers git:(master) × ./app
@value1: 0x16b16754c
@ptr1:   0x16b167540
value1:  42
ptr1:    0x16b16754c
→  26-assigning-pointers git:(master) ×
```

Dereferencing pointers (* operator)

26-assigning-pointers/main.cpp

```cpp
int main(int argc, char const *argv[]) {
  int value1 = 42;
  int* ptr1 = nullptr;
  ptr1 = &value1;
  std::cout << "value1:  " << value1 << std::endl;
  std::cout << "ptr1:    " << *ptr1 << std::endl;

  value1 = 1337;
  std::cout << "value1:  " << value1 << std::endl;
  std::cout << "ptr1:    " << *ptr1 << std::endl;

  *ptr1 = 6128;
  std::cout << "value1:  " << value1 << std::endl;
  std::cout << "ptr1:    " << *ptr1 << std::endl;
  return 0;
}
```

You can read/write the variable either by the variable itself or by its pointer.



Pointers and functions

Pointers are often used as Function parameters. This means passing the pointer to the argument variable rather than the variable itself.

A function that uses a pointer as a parameter is able to modify the pointed value (as with references)

The function have to dereference the pointer to access or change the value being pointed to.

```
void foo(int* y) {
    std::cout << "Inside foo, y = " << *y << std::endl;
    (*y)++;
    std::cout << "After update inside foo, y = " << *y << std::endl;
}
```

28-passing-by-pointer/main.cpp

```cpp
void foo(int* y) {
    std::cout << "   Inside foo, y = " << *y << std::endl;
    (*y)++;
    std::cout << "   After update inside foo, y = " << *y << std::endl;
}
int main(int argc, char const *argv[]) {
    int x = 5;
    int* z = &x; // create a pointer
    std::cout << "Inside main, before foo, x = " << x << std::endl;
    std::cout << "Inside main, before foo, z = " << *z << std::endl;
    foo(z); // Call foo with the pointer z
    std::cout << "Inside main, after foo, x = "
              << x << std::endl;
    std::cout << "Inside main, after foo, z = "
              << *z << std::endl;
    foo(&x); // Call foo with the address of x
    std::cout << "Inside main, after foo, x = "
              << x << std::endl;
    std::cout << "Inside main, after foo, z = "
              << *z << std::endl;
    return 0;
}
```

```
-zsh — d0m                                    ⌥⌘1
→  28-passing-by-pointer git:(master) ✗ make
clang++ -Wall -std=c++20  -MMD -c  main.cpp
clang++ -Wall -std=c++20   -o app main.o
→  28-passing-by-pointer git:(master) ✗ ./app
Inside main, before foo, x = 5
Inside main, before foo, z = 5
   Inside foo, y = 5
   After update inside foo, y = 6
Inside main, after foo, x = 6
Inside main, after foo, z = 6
   Inside foo, y = 6
   After update inside foo, y = 7
Inside main, after foo, x = 7
Inside main, after foo, z = 7
→  28-passing-by-pointer git:(master) ✗
```

The next() function

29-Date-passing-by-pointer/date.h

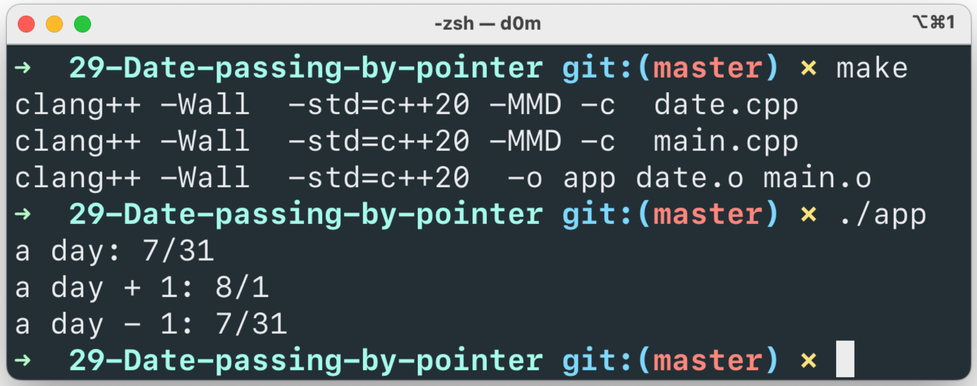
```
void next(Date* d);
```

29-Date-passing-by-pointer/date.cpp

```
void next(Date* d) {
    if ((d->month()==12) && (d->day()==31)) {
        d->updateDay(1);
        d->updateMonth(1);
    }
    else if (d->day()==getDaysInMonth(d->month())) {
        d->updateDay(1);
        d->updateMonth(d->month()+1);
    }
    else {
        d->updateDay(d->day()+1);
    }
}
```

Use a pointer each time we call a Date object as an argument of a function, i.e for the next() and back() functions

Use -> operator when dealing with a pointer in the body of a function, e.g. d->month() instead of (*d).month()



References

```
void foo(int& y) {
  y++;
  std::cout << y << std::endl;
}
```

1. References must be initialized when created
2. Once initialized, references cannot be reinitialized
3. References can never be NULL
4. Reference is automatically dereferenced

VS

1. Pointers can be created and initialized any time
2. Pointers can be reinitialized any number of times
3. Pointers can be nullptr
4. * is explicitly used to dereference a pointer

```
void foo(int* y) {
  (*y)++;
  std::cout << *y << std::endl;
}
```

Pointers

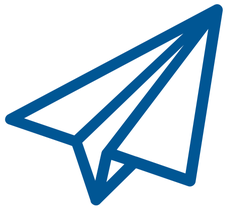
Photo by JESHOOTS.COM on Unsplash

RULES #06

When should I use References, and when should I use Pointers?

1. References are generally preferred to pointers in C++ because pointers look like an obsolete mechanism inherited from C.
2. Use References when you can, and pointers when you have to (no other choice).

A THIRD TAKE HOME MESSAGE ABOUT PASSING ARGUMENTS



#3

C++ provides different ways to pass arguments to a function.

1. If the object passed to the function has to be updated, preferably call by reference or use a pointer.
2. If the object passed to the function is only read and if this object is big, call by const reference.
3. Otherwise, call by value.

Questions

AGENDA

01 – Basics of Objects / Classes

02 – A realistic example of class

03 – More on functions

04 – Other user-defined types

05 – Organize your types in namespaces

Other C++ user-defined types

In addition to classes, C++ offers the possibility to create other user-defined types.

STRUCTURES

PUBLIC CLASSES

Used mainly for plain old data.
Inherited from C.

ENUMERATION

RANGE OF VALUES

Used for variables that can only take one value out of a set of possible values.

UNION

SPECIAL CLASS TYPE

Used to save memory by holding only one data at a time.

A first example of struct

Struct = Public class

Used mainly for plain old data (i.e. a bundle that just stores data with little logic).

Default access is public for backwards compatibility with C whereas default access is private for class.

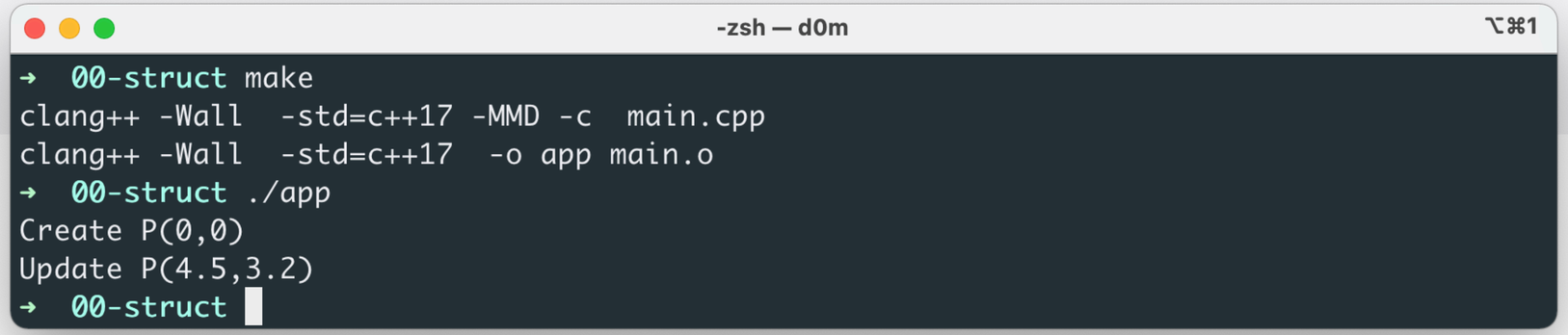
point.h

```
#ifndef POINT_H
#define POINT_H

struct Point {
    float x; // x and y are public
    float y; // No need to write getters/setters
  };
#endif // POINT_H
```

Using the struct Point

main.cpp

```cpp
#include <iostream>
#include "point.h"

int main(int argc, char const *argv[]) {
    // C++ declaration - In C, we must declare: struct Point p1;
    Point p1; // zero-initialization
    std::cout << “Create P(" << p1.x << "," << p1.y << ")"<< '\n';
    p1.x = 4.5; // x is public
    p1.y = 3.2; // x is public
    std::cout << "Update P(" << p1.x << "," << p1.y << ")"<< '\n';
    return 0;
}
```



struct Point is a class

Struct can also have constructors and functions as standard classes.

No real difference between class and structure: a structure can be viewed as a public class!

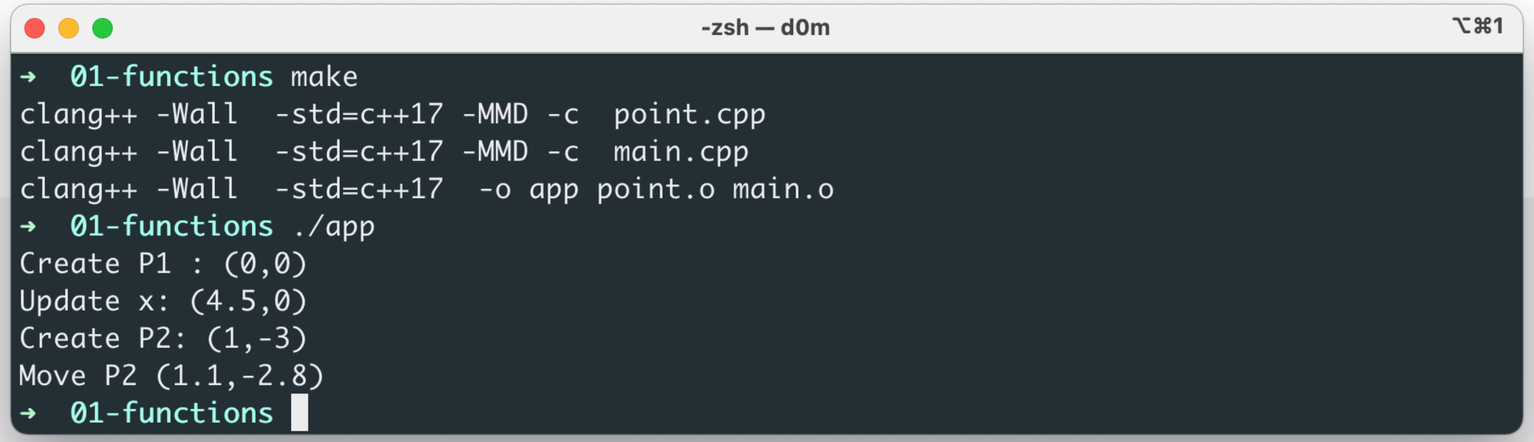
point.h

```
struct Point {
    float x; // x and y are public
    float y; // No getters/setters
    Point(float x=0.0, float y=0.0);
    void move(float dx, float dy);
};
```

point.cpp

```
#include "point.h"
Point::Point(float x, float y) : x(x), y(y) {}
void Point::move(float dx, float dy) {
    x+= dx;
    y+= dy;
}
```

struct Point is a class



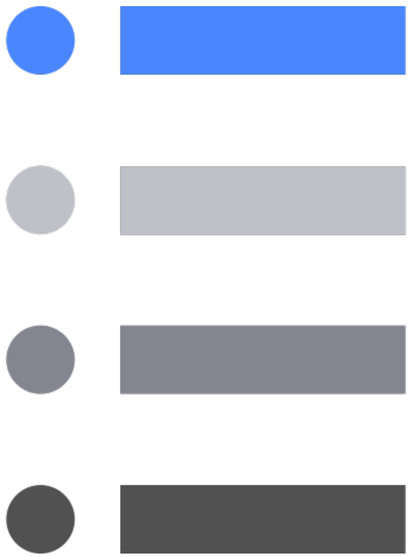
main.cpp

```cpp
#include "point.h"

int main(int argc, char const *argv[]) {
    Point p1; // Initialized to default values from constructor
    std::cout << "Create P1 : (" << p1.x << "," << p1.y << ")"<< std::endl;
    p1.x = 4.5;
    std::cout << "Update x: (" << p1.x << "," << p1.y << ")"<< std::endl;
    Point p2(1.0, -3.0); // declaration only valid in C++
    std::cout << "Create P2: (" << p2.x << "," << p2.y << ")"<< std::endl;
    p2.move(0.1, 0.2); // call of the move function
    std::cout << "Move P2 (" << p2.x << "," << p2.y << ")"<< std::endl;
    return 0;
}
```

A first example of enum

enum = range of predefined values

User-defined data type which can be assigned a set of unique values that are defined by the programmer at the time of declaring the enumerated type.



status.h

```h
#ifndef STATUS_H
#define STATUS_H
enum Status { // Braces surround the entries
    Pending,   // a comma-separated
    Urgent,    // set of constants
    Delayed,   // (integers) with unique names
    Cancelled,
    Done       // No comma after the last one
}; // Do not forget the semi-colon
#endif // STATUS_H
```

A first example of enum

main.cpp

```cpp
#include <iostream>
#include "status.h"
int main(int argc, char const *argv[]) {
    Status status; // or enum Status status;
    status = Delayed;
    // implicit cast of status into integer
    std::cout << "Delayed: " << status << std::endl;
    status = Urgent;
    std::cout << "Urgent: " << status << std::endl;
    return 0;
}
```

status.h

```cpp
enum Status {
    Pending,
    Urgent,
    Delayed,
    Cancelled,
    Done
};
```

```
-zsh — d0m                                                    ⌥⌘1
➜  00-enum make
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app main.o
➜  00-enum ./app
Delayed: 2
Urgent: 1
➜  00-enum
```

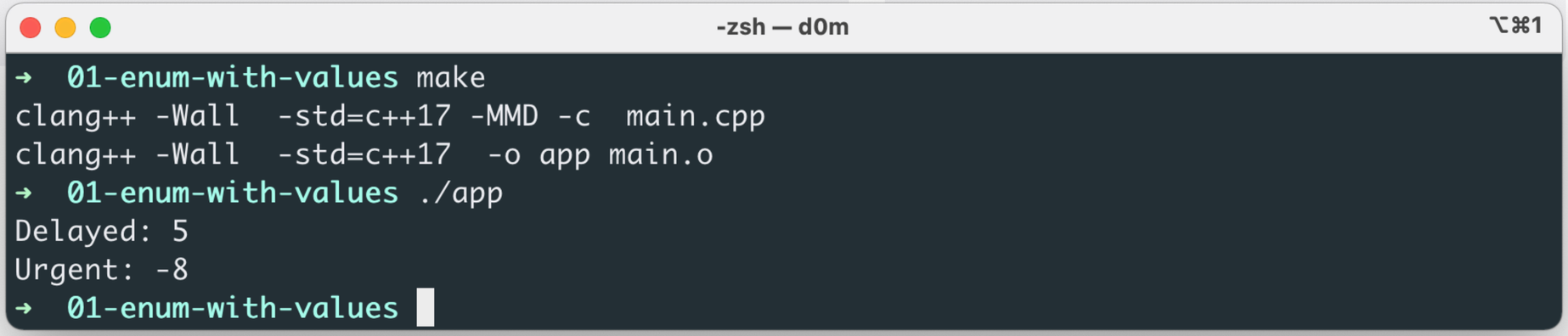
A second example of enum

main.cpp

```cpp
#include <iostream>
#include "status.h"
int main(int argc, char const *argv[]) {
    Status status; // or enum Status status;
    status = Delayed;
    // implicit cast of status into integer
    std::cout << "Delayed: " << status << std::endl;
    status = Urgent;
    std::cout << "Urgent: " << status << std::endl;
    return 0;
}
```

status.h

```cpp
enum Status {
    Pending = 12,
    Urgent = -8,
    Delayed = 5,
    Cancelled = 9,
    Done = 4
};
```



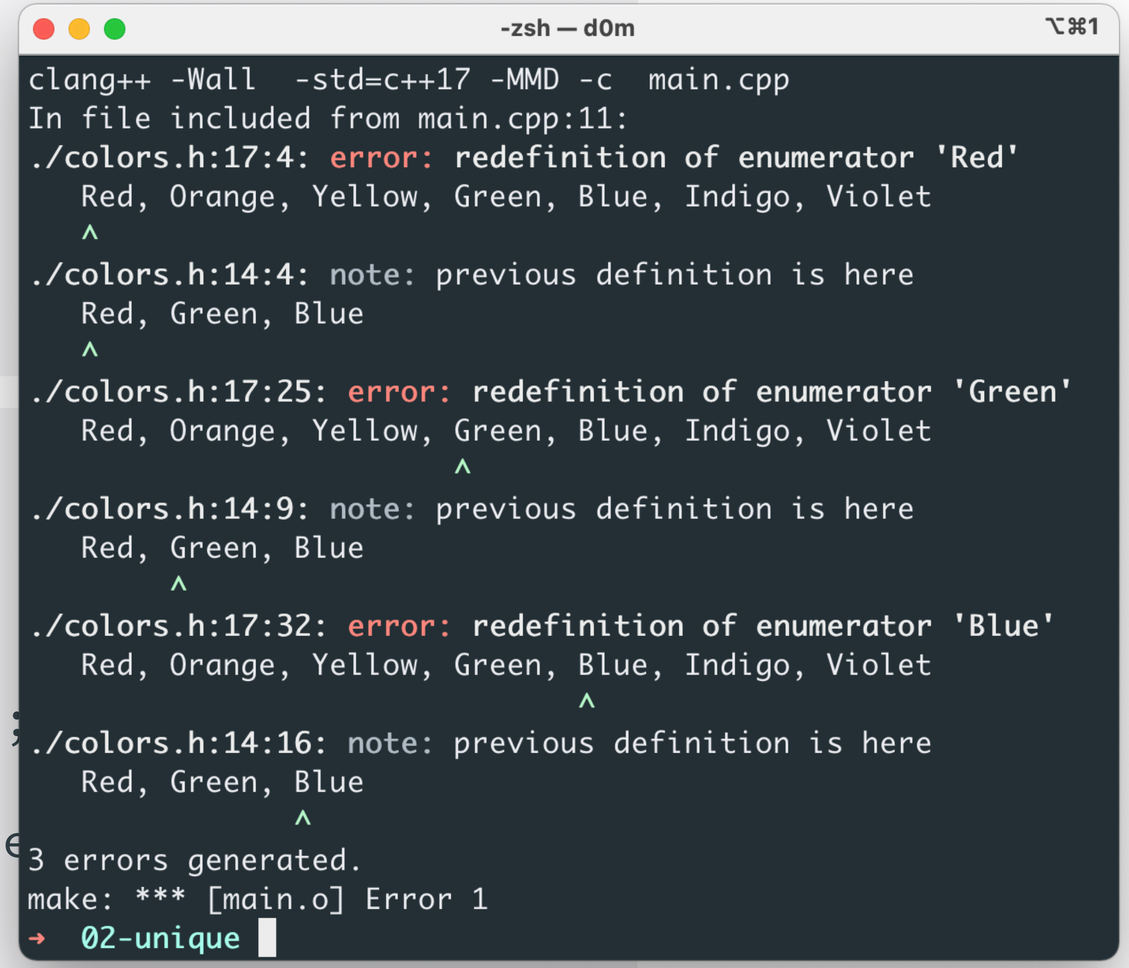
Uniqueness of values

colors.h

```
#ifndef COLORS_H
#define COLORS_H
enum  RGB { Red, Green, Blue };
enum  ROYGBIV { Red, Orange, Yellow, Green,
Blue, Indigo, Violet };
#endif // COLORS_H
```

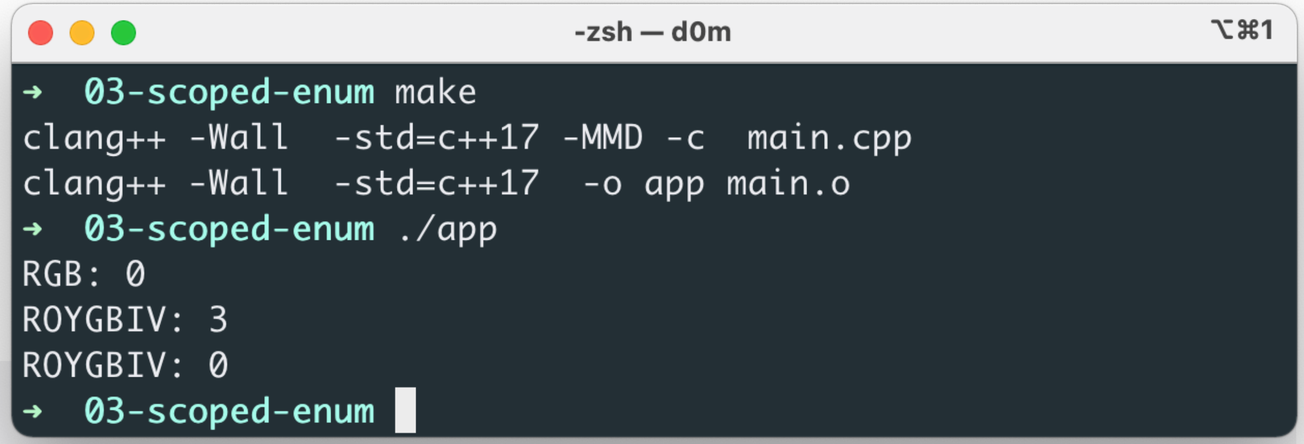
main.cpp

```
#include "colors.h"
int main(int argc, char const *argv[]) {
    RGB color1 = Red;
    std::cout << "RGB: " << color1 << std::endl;
    ROYGBIV color2 = Violet ;
    std::cout << "ROYGBIV: " << color2 << std::e
    return 0;
}
```



Scoped enum

C++11 has introduced enum classes (also called scoped enum), that makes enumerations both strongly typed and strongly scoped.



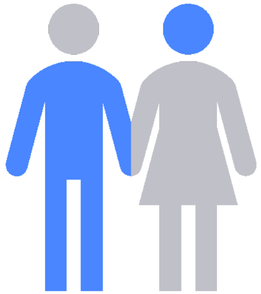
colors.h

```h
#ifndef COLORS_H
#define COLORS_H

enum class RGB {
    Red, Green, Blue };
enum class ROYGBIV {
    Red, Orange, Yellow,
    Green, Blue, Indigo,
    Violet };
#endif // COLORS_H
```

main.cpp

```cpp
int main(int argc, char const *argv[]) {
    // Use fully qualified name
    RGB color1 = RGB::Red;
    std::cout << "RGB: " << static_cast<int>(color1);
    ROYGBIV color2 = ROYGBIV::Green ;
    std::cout << "ROYGBIV: " << static_cast<int>(color2);
    ROYGBIV color3 = ROYGBIV::Red ;
    std::cout << "ROYGBIV: " << static_cast<int>(color3);
    return 0;
}
```

A first example of union



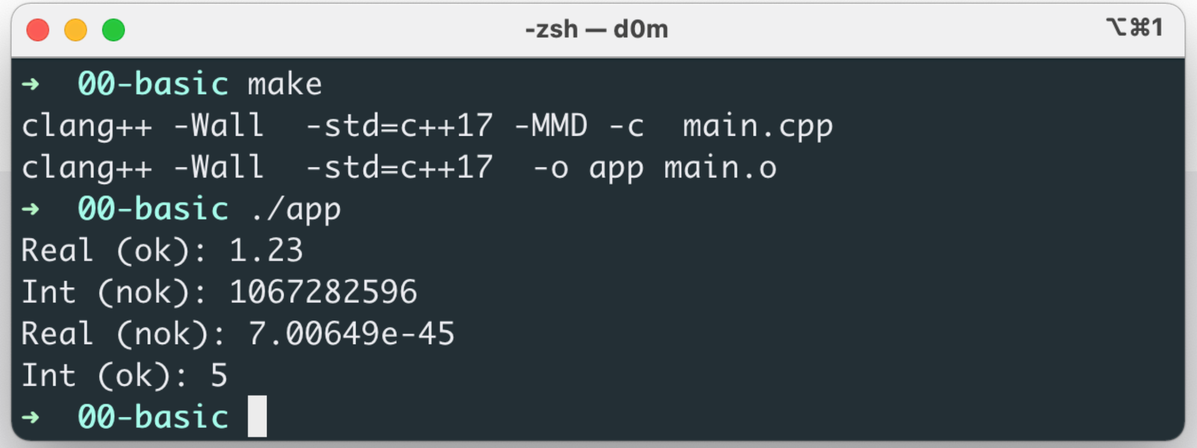
Union = Special class type

Used to save memory by holding only one data at a time.

number.h

```
#ifndef NUMBER_H
#define NUMBER_H

union Number {          // The purpose of union is to save memory
      float real;       // by using the same memory region for storing
      int integer;      // different objects at different times.
};

#endif // NUMBER_H
```

A first example of union



main.cpp

```cpp
#include <iostream>
#include "number.h"

int main(int argc, char const *argv[]) {
    Number nb;
    nb.real = 1.23;
    std::cout << "Real (ok): " << nb.real << std::endl;
    std::cout << "Int (nok): " << nb.integer << std::endl;
    nb.integer=5;
    std::cout << "Real (nok): " << nb.real << std::endl;
    std::cout << "Int (ok): " << nb.integer << std::endl;
    return 0;
}
```

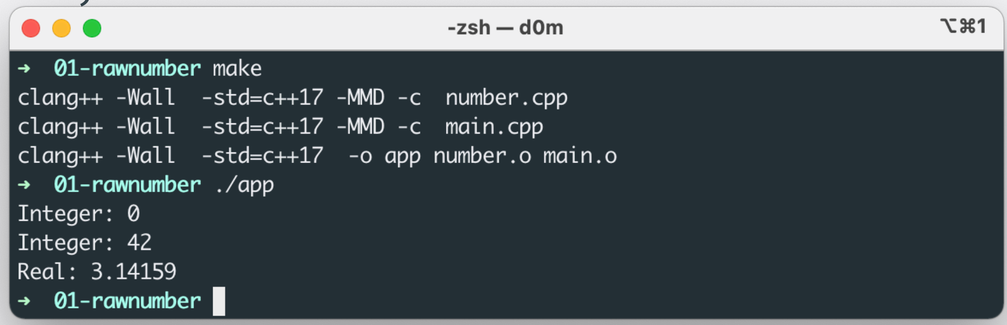
A more complex example of union

number.h

```
union Number {
	int integer;
	float real;
};
enum class Type {
	integer, real
};
class Rawnumber {
public:
	Rawnumber(int number=0);
	Rawnumber(float number);
	void display();
private:
	Type _type;
	Number _number;
};
```

main.cpp

```
#include "number.h"

int main(int argc, char const *argv[]) {
    Rawnumber nb1;
    nb1.display();
    int int_nb = 42;
    Rawnumber nb2(int_nb);
    nb2.display();
    float real_nb = 3.14159;
    Rawnumber nb3(real_nb);
    nb3.display();
    return 0;
}
```



Questions

AGENDA

01 – Basics of Objects / Classes

02 – A realistic example of class

03 – More on functions

04 – Other user-defined types

05 – Organize your types in namespaces

USING NAMESPACES

Namespaces in C++ are used to organize code into logical groups and to prevent name collisions that can occur with large projects.

Photo by Damir Kopezhanov on Unsplash

Namespaces

A typical situation

Consider a situation, we are writing a function called abc() and there is another predefined library with the same function abc().

Now at the time of compilation, the compiler has no clue which version of abc() function we are referring to within our code.

The need for Namespaces

Namespace are used to organize and differentiate similar functions, variables, classes, etc. with the same name.

Using namespace, we can define the context (i.e. scope) in which names are defined.

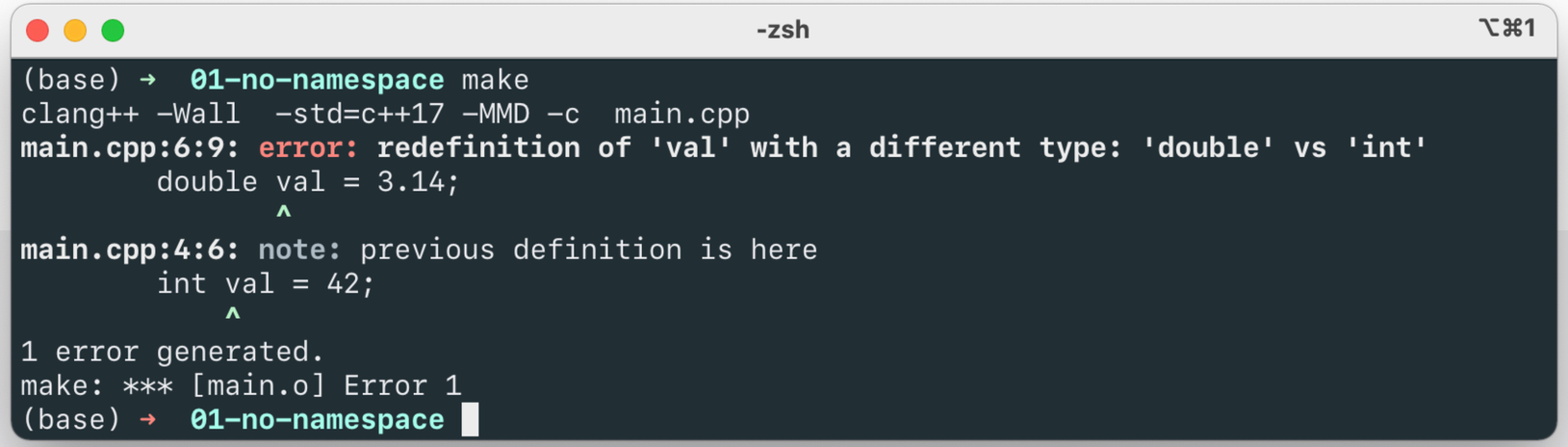
The std namespace

All C++ standard library types and functions are declared in the std namespace or namespaces nested inside std thus it is widely used in most of the programs.

Introduction to namespace

In each scope, a name can only represent one entity.

So, there cannot be two variables with the same name in the same scope.



main.cpp

```cpp
int main() {
    int val = 42;
    std::cout << “The answer: " << val << std::endl;
    double val = 3.14;
    std::cout << “pi: " << val << std::endl;
    return 0;
}
```

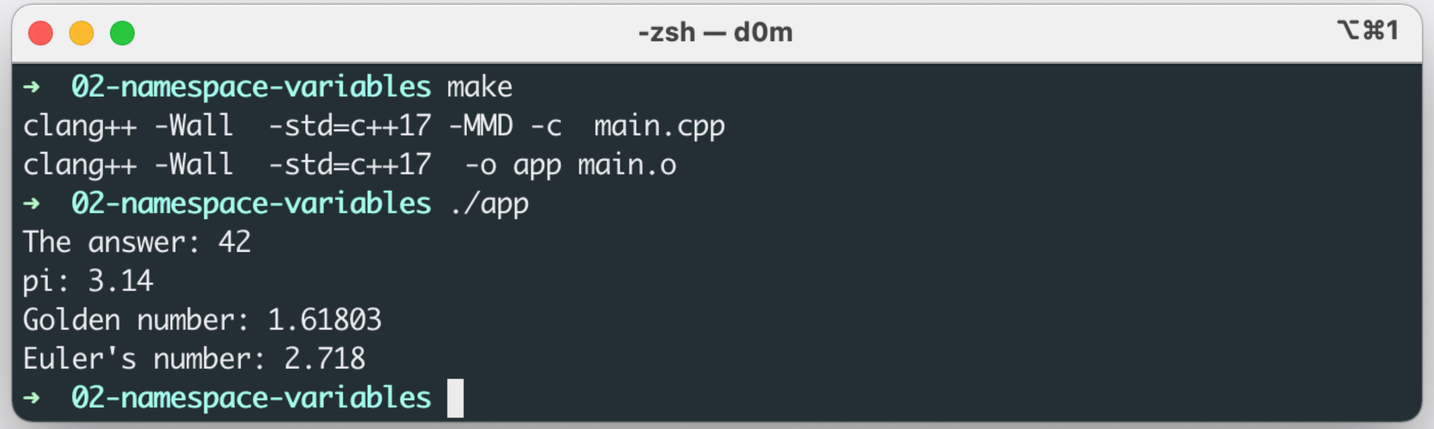
A “bad” example of namespaces

A namespace is a container for identifiers.

It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

main.cpp

```cpp
namespace ns1 {
    int val = 42;
}
namespace ns2 {
    double val = 3.14;
}
double val = 2.718;
int main() {
    std::cout << "The answer: " << ns1::val << std::endl;
    std::cout << "pi: " << ns2::val << std::endl;
    double val = 1.6180339887;
    std::cout << "Golden number: " << val << std::endl;
    std::cout << "Euler's number: " << ::val << std::endl;
    return 0;
}
```

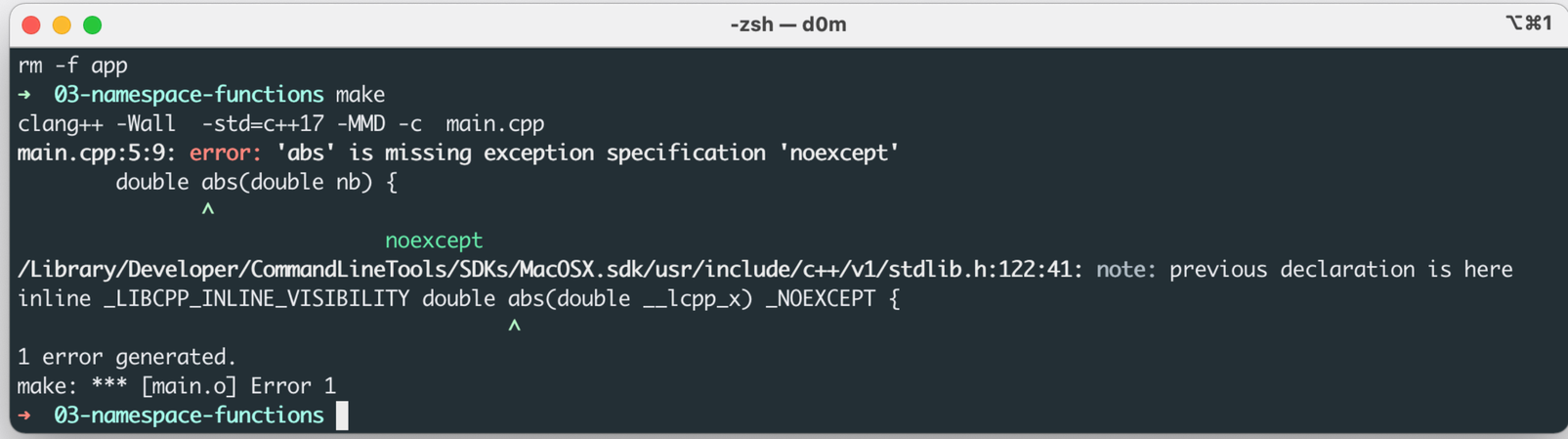


A second example of namespaces

Imagine we want to write a function that compute the absolute value of a real number.
An abs function already exists and is declared in stdlib.h.
The solution is to embed the abs function into a namespace.

main.cpp

```cpp
double abs(double nb) {
    if (nb<0) return -nb;
    return nb;
}
int main() {
    double number1 = -3.14;
    std::cout << abs(number1)
              << std::endl;
    return 0;
}
```

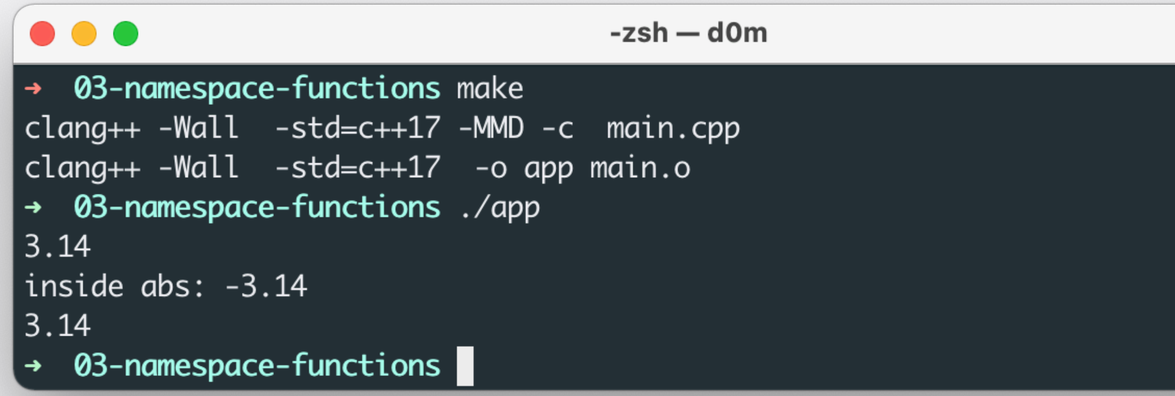


A second example of namespaces

The solution is to embed the abs function into a namespace.

main.cpp

```cpp
namespace myMath {
    double abs(double nb) {
        std::cout << "inside abs: " << nb << std::endl;
        if (nb<0) return -nb;
        return nb;
    }
}

int main() {
    double number1 = -3.14;
    std::cout << abs(number1) << std::endl;
    std::cout << myMath::abs(number1) << std::endl;
    return 0;
}
```

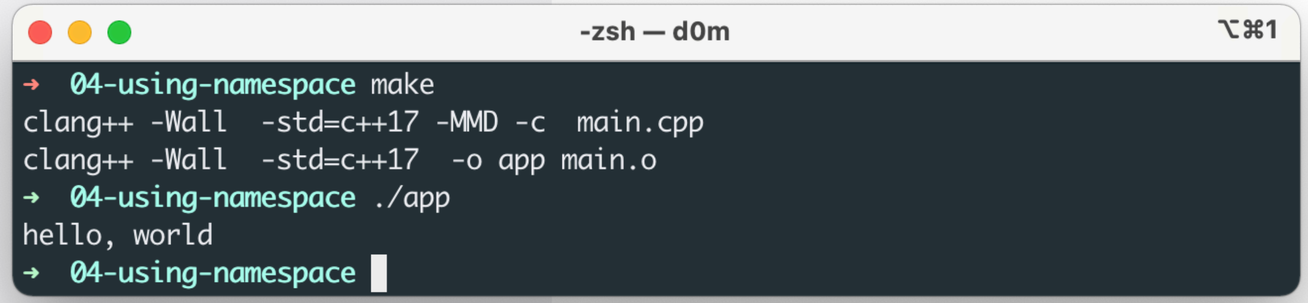


‘Using namespace” directive

To avoid writing the fully qualified name (namespace::function), you can use the “using namespace” directive.

main.cpp

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "hello, world" << endl;
    return 0;
}
```



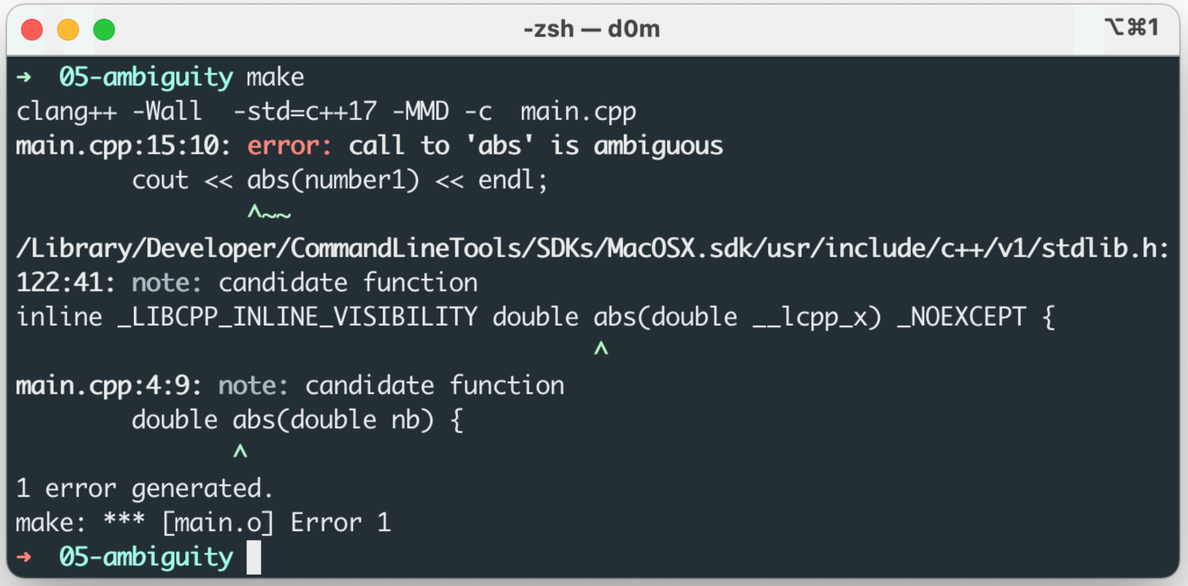
Using namespace directive is considered BAD PRACTICE.

It’s not safe because there may be ambiguity between elements from different namespaces that can have same name.

‘Using namespace” directive

main.cpp

```cpp
#include <iostream>

namespace myMath {
    double abs(double nb) {
        if (nb<0) return -nb;
        return nb;
    }
}
using namespace std;
using namespace myMath;

int main() {
    double number1 = -3.14;
    cout << abs(number1) << endl;
    return 0;
}
```



Do not use ‘using namespace’ directive. Limit to import specific identifiers with the ‘using’ directive:

```cpp
using std::cout;
using std::endl;
```

if you still want to import entire namespaces, try to do so inside limited scope and not in global scope. Do not import namespace in .h file !

SOME RULES

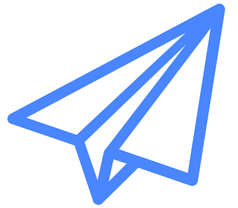
to use namespaces efficiently

Photo by Clark Tibbs on Unsplash

1. Place code in namespaces.
2. Increase readability using fully qualified names rather than using “using directives”.
3. A class and its helper functions must be defined into the same namespace.
4. Group all the user-defined types that are related to each other into the same namespace.

#4

A FOURTH TAKE HOME MESSAGE ABOUT USER TYPES



C++ mainly relies on user-defined types that are collections of data describing an object's attributes and state.

Classes are the main user-defined types and struct, enum and union are marginally employed

User-defined namespaces organize code into logical groups and prevent name collisions.

Questions

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Come visit us at

https://github.com/dginhac/esirem-itc313